UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Personalización de perfiles de notificación de aplicaciones según el contexto del usuario

Proyecto Final de Carrera

[Ingeniería Técnica Informática de Sistemas]

Autor: Nacho López Guerrero

Director: Vicente Pelechano Ferragud y Miriam Gil Pascual

Septiembre 2012

Resumen

En los últimos años los teléfonos móviles inteligentes (smartphones) se han convertido en un objeto indispensable en nuestras vidas cotidianas, esto ha provocado la aparición de todo un gran ecosistema de potentes sistemas operativos móviles, como iOS, Android o Windows Phone 7, y un gran abanico de aplicaciones compatibles con estos sistemas capaces de satisfacer todas las necesidades del usuario, tanto a nivel de ocio como de productividad. Estas aplicaciones se comunican con el usuario cuando tienen que enviar algún aviso (por ejemplo cuando llega un mail, actualización de una aplicación…) enviando una serie de notificaciones para que éste se dé por avisado. Cada sistema operativo envía estas notificaciones según unos mecanismos propios, algunos de ellos son similares, mientras que otros son particulares de cada sistema (por ejemplo una notificación “toast" en Android, o una notificación desplegable en la barra de estado de iOS). Durante este proyecto trataremos de administrar la llegada de estas notificaciones por medio de una aplicación.

La aplicación desarrollada permite al usuario administrar la manera de cómo le llegan dichas notificaciones gracias a la creación de una serie de perfiles que tendrán cada uno un grado de molestia diferente, alertando al usuario en mayor o menor medida. Cada perfil se asociará a una o más aplicaciones (servicios) cuyas notificaciones queramos personalizar.

Con esta aplicación se pretende dejar al usuario configurar el modo y nivel de molestia de las notificaciones de las diferentes aplicaciones más allá de lo que permite por defecto el sistema operativo del iPhone: iOS.

Tabla de contenidos

Índice:

Tabla de figuras

Índice:

1. Introducción

1.1 Contexto actual de sistemas operativos móviles y aplicaciones

En los últimos 4 años el mundo de la telefonía móvil ha sufrido una auténtica revolución: los nuevos teléfonos inteligentes (smartphones) han sustituido casi por completo los antiguos terminales móviles con los que solo se podía realizar una serie de tareas básicas como llamar, enviar mensajes, hacer fotografías y vídeos… Estos nuevos smartphones utilizan sistemas operativos propios casi tan potentes como pueden ser los sistemas operativos de sobremesa (con arquitectura x86). Estos sistemas corren sobre unos avanzados procesadores multinúcleo (generalmente con arquitectura ARM) que hacen que los teléfonos sean tan potentes y productivos como pequeños ordenadores de bolsillo, permitiendo al usuario realizar una serie de tareas hasta entonces impensables en un teléfono, como navegar por internet, recibir e-mail, utilizar GPS, grabar vídeos en alta definición… Y todo ello manejando el teléfono con nuestros dedos, gracias a precisas pantallas capacitivas y a software que permite que la interfaz de usuario (UI) responda a los toques de manera fluida y sin retardo.

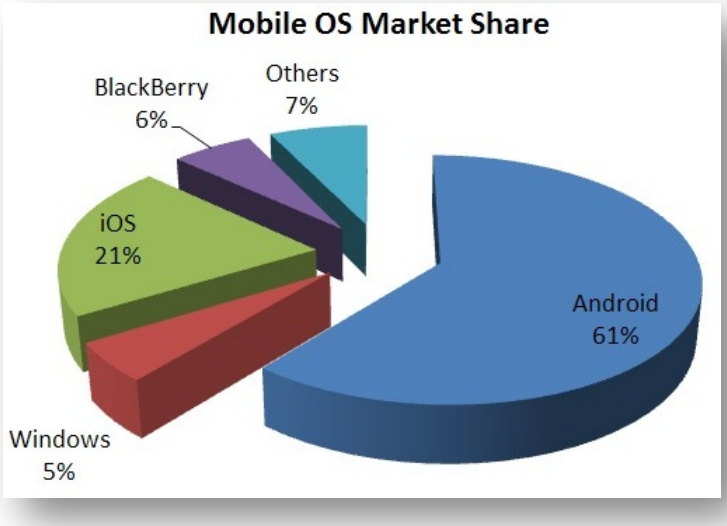


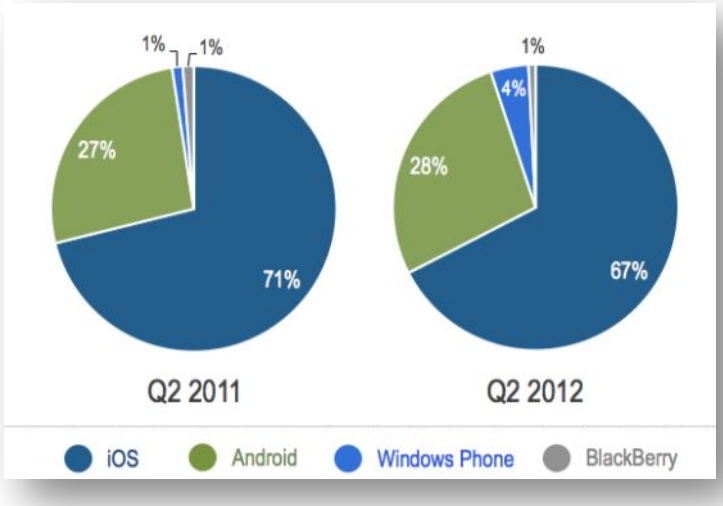
Figura 1. Cuota de mercado de SOs móviles en 2012

Esta proliferación del mercado de telefonía móvil ha ocasionado la creación de potentes sistemas operativos móviles, como iOS, Android, BlackBerry OS, el casi desaparecido Symbian o el reciente Windows Phone 7 que cada vez va tomando más importancia en el mercado. A las funcionalidades que nos permiten realizar estos sistemas operativos (SOs) por defecto, se la han añadido una infinidad de **nuevas aplicaciones** capaces de extender la funcionalidad de dichos sistemas más allá de lo imaginable. Ha aparecido un mercado de aplicaciones móviles donde cualquier desarrollador puede publicar una aplicación que él haya implementado en las llamadas 'tiendas de aplicaciones'. Poniendo así su trabajo a disposición de millones de personas, reduciendo el número de intermediarios necesarios para que la

aplicación llegue desde el creador hasta el consumidor final. Esto conlleva un aumento de los ingresos que dicho desarrollador obtiene por la venta de su aplicación, siendo un negocio muy rentable actualmente. Este mercado de aplicaciones móviles cubre ya un gran abanico de necesidades que el usuario puede tener a la hora de utilizar su terminal, tanto a nivel profesional: aplicaciones de productividad, diseño/desarrollo software, aplicaciones de contabilidad, etc. como a nivel de ocio: videojuegos, reproductores MP3, reproductores de vídeo, aplicaciones para acceder a redes sociales... En la parte superior podemos ver una imagen (Figura 1) donde se muestra la cuota de mercado de cada SO (según la International Data Corporation (IDC)). Podemos observar que Android domina en este aspecto, esto es debido a que es un sistema operativo libre que puede funcionar sobre gran variedad de hardware (y con ello fabricantes) gracias a que las aplicaciones están escritas en lenguaje de programación Java, y este se ejecuta sobre una máquina virtual Dalvik (en el caso particular de Android) "emulando" una arquitectura que hace posible esta diversidad [1]. En iOS (y en todo el ecosistema de Apple) por otra parte se utiliza el lenguaje de programación Objective C, del que se hablará de manera extendida dentro del capítulo 2 de esta memoria.

Figura 2. Proyectos empezados entre Q2 2011 y Q2 2012

Este mercado está dando lugar a otro mercado emergente muy relacionado con el anterior, ya que comparten sistemas operativos y algunas aplicaciones entre terminales, hablamos del mercado de las tabletas multitáctiles o tablets. Este mercado surgió a raíz de la creación del iPad, una tableta creada por la empresa Apple, y a día de hoy sigue creciendo con las tabletas Android, el RIM PlayBook y con las próximas tabletas con Windows 8 (próximo SO de Microsoft), convirtiéndose día tras día en el nuevo referente de la informática portátil, amenazando el mercado de ordenadores portátiles.



Esta nueva línea de productos favorece la aparición de otro mercado paralelo: el de venta y distribución de aplicaciones para SOs móviles. En la captura superior (Figura 2) podemos observar el número de nuevos proyectos de aplicaciones que se empezaron para

cada uno de los principales SO móviles entre el segundo cuarto de año de 2011 y 2012, según la consultora Flurry Analytics, como podemos observar el grueso de aplicaciones corresponde a los dos sistemas más influyentes actualmente: iOS y Android; también podemos observar como poco a poco Microsoft va tomando partido con su nuevo Windows Phone 7. La aplicación que se detalla en la memoria de este proyecto encaja perfectamente en este mercado, pasaremos a describir su utilidad brevemente en el siguiente apartado.

1.2 Tareas que realiza nuestra aplicación

Los sistemas operativos móviles ponen a disposición del desarrollador unos mecanismos de interacción para notificar al usuario cada vez que una aplicación así lo requiera, por ejemplo mostrando un mensaje (dialog) en la pantalla, un texto en la barra de notificaciones, un diodo led de notificaciones que se ilumina cuando llega un aviso, una notificación dirigida al centro de notificaciones o los medios más clásicos como son el sonido o la vibración. Se puede observar que cada mecanismo de interacción supone un aviso más o menos intrusivo de cara al usuario haciendo que la notificación entrante llame más o menos la atención de éste dependiendo del mecanismo empleado.

El objetivo de nuestra aplicación es permitir al usuario administrar la manera en la que estas notificaciones lo avisan, pudiendo así configurar el grado de intrusión o molestia de dichas notificaciones. Para ello, se crean una serie de grados de molestia (perfiles) configurables por el usuario, que serán asociados a las aplicaciones cuya notificaciones éste quiera controlar. Cada uno de estos perfiles de molestia tendrá asociados una serie de mecanismos de interacción que definirán el modo y nivel de molestia de las notificaciones enviadas por dichas aplicaciones. De este modo, todas las notificaciones entrantes de una aplicación serán enviadas según el filtro definido por su perfil asociado, “molestando” al usuario en la medida que éste desee. Además, también podremos ajustar este comportamiento de una manera más avanzada: pudiendo definir transiciones entre los perfiles, haciendo que cuando estemos ejecutando una aplicación con un perfil asociado y se cumplan una serie de condiciones, las notificaciones de dicha aplicación pasen a enviarse según el filtro definido por otro perfil de molestia. El usuario podrá configurar todo esto a su antojo, seleccionando qué aplicaciones de las que tengan ese perfil asociado se verán afectadas y pudiendo definir las condiciones que provocarán la transición entre perfiles según sus necesidades.

Esta aplicación nos permitirá extender y personalizar el comportamiento de notificación por defecto del sistema operativo iOS, siendo algo muy útil ya que cada vez el tipo de usuario del terminal es más amplio y con ello aumenta el número de diferentes necesidades de notificación que pueda tener cada uno. Prueba de ello es que en la última versión de Android, Jelly Bean (4.1), se está empezando a implementar un sistema parecido al que propone esta aplicación, siendo solo accesible (de momento) a desarrolladores, y del cual añadimos unas capturas continuación:

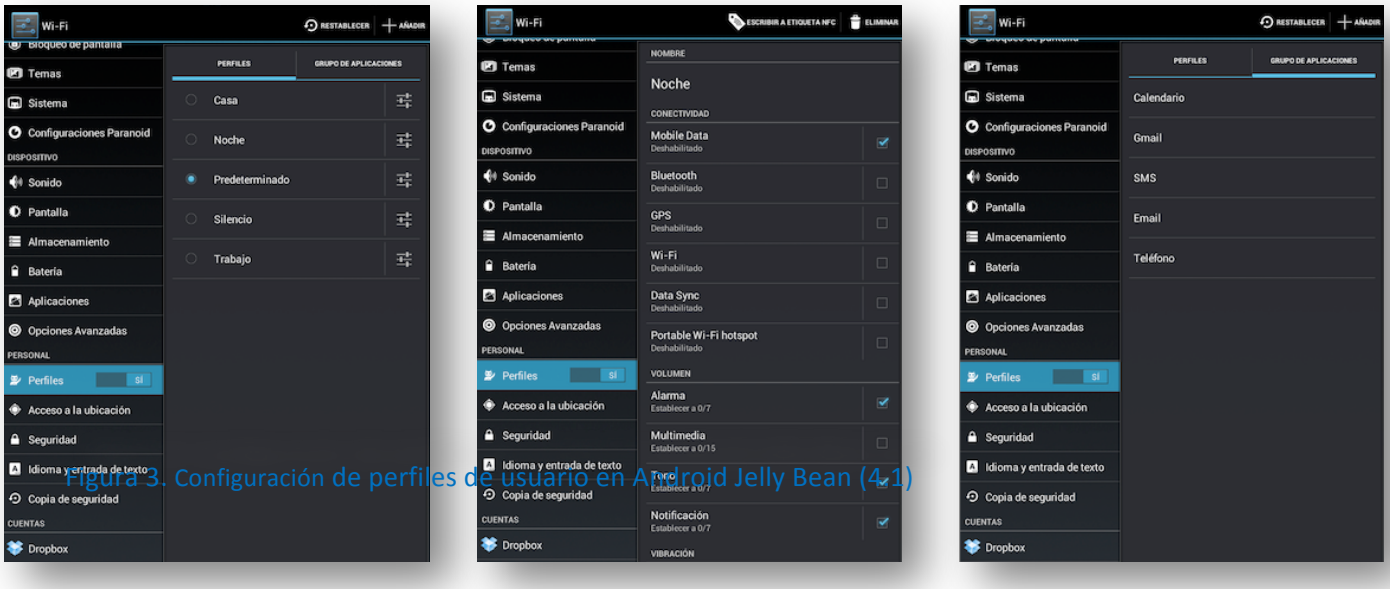


Figura 3. Configuración de perfiles de usuario en Android Jelly Bean (4.1)

1.3 Estructura de la memoria del proyecto

En este apartado pasamos a describir como se va a estructurar esta memoria del proyecto fin de carrera titulado: "Adaptación de aplicaciones móviles al comportamiento de usuario". La memoria constará de 5 capítulos, cuyo contenido se expone a continuación:

- **Capítulo 1:** Es el capítulo actual, en el que hemos comentado la situación actual del mercado de terminales, de la venta de aplicaciones para ellos y para finalizar hemos descrito de manera rápida la misión de nuestra aplicación dentro de todo este contexto.
- **Capítulo 2:** En el segundo capítulo analizaremos los frameworks y servicios que nos ofrece Apple dentro del SDK de iOS para hacer posible el desarrollo de aplicaciones. También se analizarán las principales características del lenguaje de programación utilizado para ello, Objective C.
- **Capítulo 3:** En este tercer capítulo ampliaremos la información sobre la aplicación creada, ofreciendo una visión ampliada de su funcionalidad así como de la arquitectura multicapa utilizada llevar a cabo su implementación.
- **Capítulo 4:** En este capítulo se ofrece un ejemplo de un escenario de uso de la aplicación en la vida real, creando una serie de transiciones para unos determinados servicios, que se darán según las condiciones creadas a partir de las necesidades del usuario.
- **Capítulo 5:** En este último capitulo se analizarán las conclusiones sobre el proyecto y se propondrán ampliaciones que se podrían aplicar en un futuro para mejorar la aplicación.

2. Lenguaje y herramientas para el desarrollo en iOS

En este segundo capítulo analizamos todo lo necesario para hacer posible la programación en iOS: la división por capas de software de iOS, el lenguaje de programación Objective-C y el entorno de programación Xcode, que se utiliza en todo el ecosistema Apple.

2.1 Desarrollo en iPhone: iOS

La aplicación que hemos desarrollado se ha hecho teniendo en mente la versión 5.0 del sistema operativo del iPhone, iOS, cuyo código fuente (SDK) pone Apple a disposición de los desarrolladores de manera gratuita con el único requisito de tener un ordenador Mac con el entorno de programación Xcode instalado (que se puede descargar de manera gratuita desde el AppStore). Si el desarrollador decide publicar sus aplicaciones en la AppStore deberá pagar por una licencia de desarrollador, que tiene un precio de 99$ (unos 80€) [2] y una validez de un periodo de un año.

iOS es un sistema operativo que surgió con la salida del primer iPhone en 2007 (inicialmente llamado iPhone OS). Sus aplicaciones se escriben en el lenguaje de programación Objective-C, cuyo núcleo está basado en Darwin BSD [3] (al igual que Mac OSX). Supuso una auténtica revolución el mundo de la telefonía móvil y la informática en general. Aunque tenía una serie de limitaciones inicialmente -como la ausencia de la función de cortar/pegar, de centro de notificaciones, de multitarea...- a día de hoy, gracias a las constantes actualizaciones por parte de Apple durante ya 5 años, la mayoría de éstas deficiencias están ya resueltas. La última versión de este sistema operativo es la versión 6.0 que se liberó al público hace muy poco, en Septiembre de 2012, siendo totalmente compatible nuestra aplicación con esta versión.

A continuación exponemos las principales novedades que incluye la versión 5.0 de iOS, en la que su principal novedad fue la incorporación de un centro de notificaciones: un sitio donde agrupar todos los avisos que envían las diferentes aplicaciones.

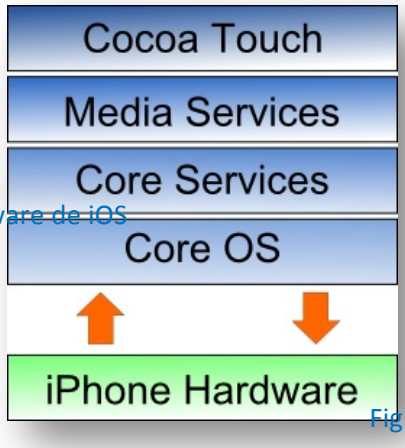
- Notification Center
- Mejoras de notificaciones de las aplicaciones
- Lockscreen notifications
- Opciones de notificación para cada aplicación
- Notificaciones clasificadas
- iMessage
- Newstand
- Reminders

- Integración Twitter
- Integración iCloud
- Abrir camera.app en el lockscreen
- Lista de lectura de Safari
- Mejoras del rendimiento
- Navegación privada
- Sincronización vía Wi-Fi
- BackUp y Restore desde iCloud
- Foto de perfil para Game Center
- AppStore se integra con Game Center para la descarga de apps.
- Nuevos logros en Game Center
- Gestos de multitarea
- División del teclado en iPad para mas comodidad
- Emoticonos
- LED Flash de notificaciones
- Vibración personalizada
- Uso de espacio
- Uso de iCloud
- Se puede activar o desactivar las “badges” de los iconos
- FaceTime en 3G
- Nitro JavaScript para Safari

Uno de los mayores retos a los que se enfrenta un desarrollador si quiere programar aplicaciones para iOS es el aprendizaje del lenguaje de programación en el que se escriben dichas aplicaciones, Objective-C. Esto es debido a que su sintaxis es diferente a la del resto de lenguajes por los que un programador suele empezar, como Java, C, C++, C#, Python… Además de tener una serie de peculiaridades que lo hacen diferente al resto que hacen que sea necesario un periodo de aprendizaje para familiarizarse con el lenguaje antes de poder empezar a programar. Una vez hayamos cogido soltura suficiente podremos acceder a los diferentes frameworks y servicios que nos permite utilizar el sistema operativo, utilizando para ello las cuatro capas de software que componen iOS: Cocoa Touch, Media, Core Services y Core OS [4], cuyas librerías están escritas en Objective-C y C. Explicaremos esta arquitectura multicapa en el siguiente apartado para poner al lector en contexto y después, en los próximos apartados del capítulo, profundizaremos sobre el lenguaje de programación Objective-C y el entorno de desarrollo Xcode.

2.1.1 Arquitectura multicapa de iOS

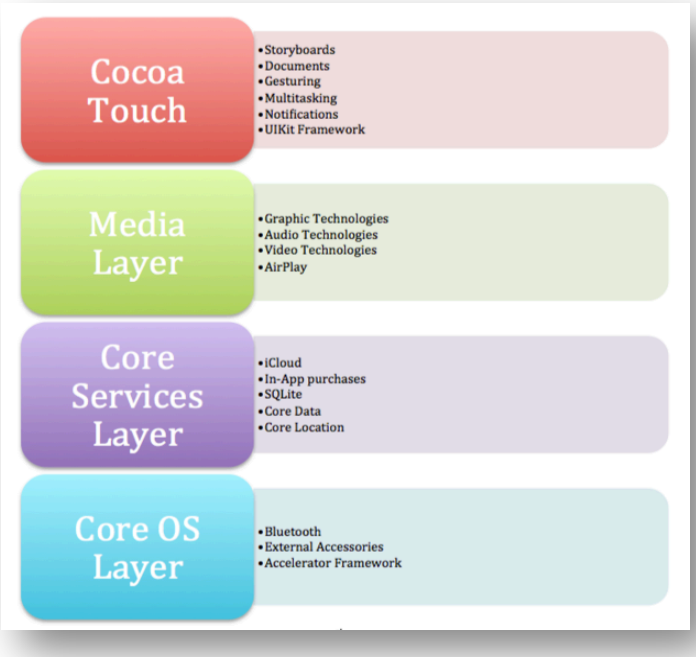
Cuando desarrollamos una aplicación para iOS no podemos acceder directamente al hardware del teléfono sino que lo tenemos que hacer a través de unas capas de software que actúan como intermediarias entre el código de nuestra aplicación y dicho hardware. Estas capas son lo que conocemos como el sistema operativo (iOS) y están formadas por unos frameworks [5] y servicios de alto nivel que hacen posible la creación de aplicaciones para iPhone. Las cuatro capas que componen el sistema operativo iOS son: Cocoa Touch, Media, Core Services y Core OS, siendo ésta última (el núcleo de iOS) la encargada de interactuar directamente con el hardware. Estos frameworks y servicios de alto nivel permiten al desarrollador interactuar con el sistema operativo y crear aplicaciones. Es preferible escribir las aplicaciones utilizando los frameworks de la capa de más alto nivel, es decir, Cocoa Touch, ya que nos permite un nivel de abstracción mayor (orientado a objetos) que permite ahorrar gran cantidad de código que sí que tendríamos que tener en cuenta en caso de utilizar las herramientas de más bajo nivel de una de las capas inferiores. La Figura 4. muestra un esquema de dichas capas.



ra 4. Capas software de iOS

Figura 5. Principales frameworks y servicios de alto nivel que nos proporcionan las diferentes capas software de iOS

Apple proporciona sus interfaces del sistema al programador en unos servicios que este puede solicitar y unos paquetes llamados frameworks, formados por librerías de enlace dinámico y todos sus recursos necesarios (cabeceras, manuales, imágenes…). Muchas de estas frameworks son compartidas entre iOS y Mac OSX, estando la principal diferencia entre estos dos sistemas en la capa destinada a la interfaz de usuario (UI), es decir Cocoa en caso de Mac OSX y Cocoa Touch en caso de iOS, ya que una esta preparada para reaccionar a la entrada de teclado y ratón y la otra a la entrada de toques a la pantalla táctil. La imagen de la derecha, Figura 5, muestra los frameworks y servicios más importantes de cada una de las capas en iOS. En los siguientes apartados explicaremos la misión de cada una de ellas.



2.1.1.1 Capa Cocoa Touch

La capa Cocoa Touch es la capa que contiene las principales frameworks para crear aplicaciones en iOS. Esta capa define la infraestructura básica de la aplicación y da soporte a tecnologías tan clave como la multitarea, entrada de toques en pantalla, notificaciones push y más servicios de alto nivel. Esta es la capa que más comúnmente se usa para diseñar aplicaciones. Los frameworks y servicios de alto nivel que ofrece esta capa son los siguientes:

- **Servicios de alto nivel**:
  - Auto Layout
  - Storyboards
  - Document Support
  - Multitasking
  - Printing
  - UI State Preservation
  - Apple Push Notification Service
  - Local Notifications
  - Gesture Recognizers
  - Peer-to-Peer Services
  - Standard System View Controllers
  - External Display Support

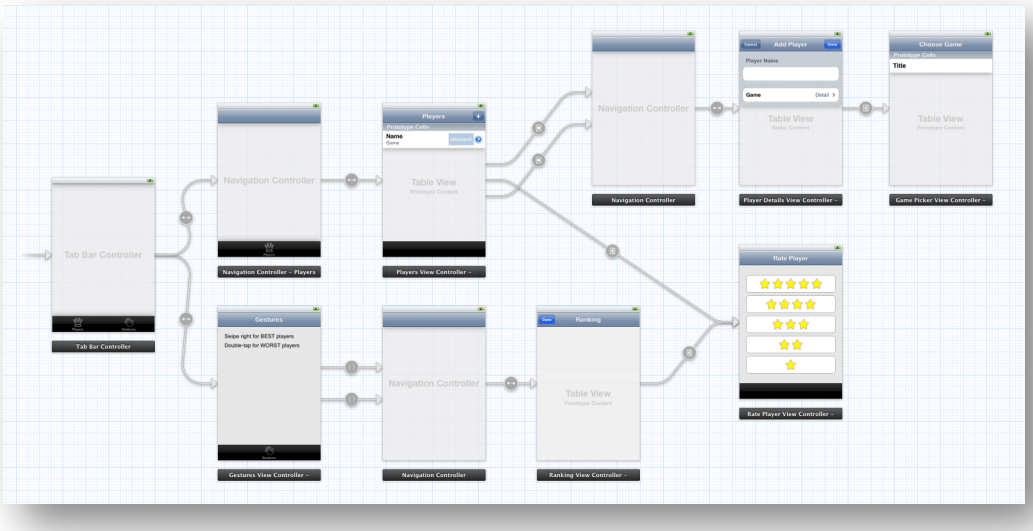
- **Frameworks**:
  - Address Book UI Framework
  - Event Kit UI Framework
  - Game Kit Framework
  - iAd Framework
  - Map Kit Framework
  - Message UI Framework
  - Twitter Framework (‘Social Framework’ desde iOS6)
  - UIKit Framework 1

A continuación ampliaremos información sobre los servicios de alto nivel y frameworks más importantes de esta capa:

Storyboards:

Servicio introducido en iOS 5 permite sustituir a los ficheros *.nib* a la hora de crear interfaces de usuario (hablaremos de ellos más adelante en la sección de Xcode). Los Storyboards permiten al desarrollador crear toda la interfaz gráfica de su aplicación en un mismo sitio, permitiendo así tener una vista más clara y gráfica de toda la interfaz de su aplicación, mostrando todas las *Views y* los *viewControllers* que la componen en un mismo lugar. Nos permite hacer de manera gráfica (con ciertas limitaciones) lo que hasta la fecha se tenía que hacer necesariamente mediante código. Podemos realizar toda nuestra aplicación en un solo Storyboard, o podemos usarlo para crear una parte de nuestra interfaz, en caso de elegir la segunda opción, podremos acceder al Storyboard creado a partir del framework *UIKitFramework*, de esta misma capa, del cual hablaremos a continuación.

Figura 6. Ejemplo de Storyboard de una App iOS en Xcode



UIKitFramework:

Esta framework es una de las más importantes que nos proporciona la API de la capa Cocoa Touch. Pone a disposición del desarrollador los mecanismos necesarios para manejar gran cantidad de los eventos de las aplicaciones en iOS tales como la multitarea, impresión de documentos, accesibilidad para discapacitados, soporte para Storyboards, cortar/copiar/pegar, soporte para animaciones del contenido de la interfaz de usuario, notificaciones locales y push, creación documentos PDF, etc. Este framework nos proporciona clases tan importantes como *UITableView,* basada en la clase *UIView*, por lo que es un framework cuyo uso es imprescindible. También nos permite controlar parte del hardware que lleva incorporado el dispositivo como el acelerómetro, la cámara, estado de la batería, sensor de proximidad o el control remoto de los auriculares conectados al teléfono.

Document support:

Es un framework disponible a partir de la versión 5.0 de iOS. En él se introdujo la clase *UIDocumet* dentro del *UIKitFramework* y utilizada para manejar los datos de la aplicación dependientes de documentos del usuario. Esta clase facilita en gran medida el manejo de documentos, especialmente aquellos alojados en el servicio en la nube de Apple, iCloud [6]. La clase *UIDocument,* nos proporciona todos los mecanismos necesarios para la lectura/escritura asíncrona de datos en ficheros (documentos), detectando automáticamente los conflictos que se puedan generar con iCloud. Para aplicaciones que usen datos del núcleo se utiliza la clase *UIManagedDocument.*

Multitasking:

La multitarea o Multitasking fue un servicio añadido a partir de la versión 4 de iOS. Fue una de las características más demandadas por los usuarios de los dispositivos iOS. Esta característica permite que una aplicación no finalice su ejecución cuando el usuario presione el botón home, sino que está seguirá ejecutándose en segundo plano (background). Se puede acceder a esta característica a partir del framework *UIKit.* Realmente, para disminuir el consumo de batería, la mayoría de aplicaciones son congeladas por el sistema al entrar en este estado de background, es decir, siguen cargadas en memoria pero sus instrucciones no se ejecutan por el procesador. A pesar de esto, el desarrollador puede hacer que la aplicación siga ejecutando ciertas instrucciones en este estado si así lo requiere. La implementación de la multitarea no requiere esfuerzo adicional por parte del desarrollador.

Gesturing:

Este servicio de alto nivel, introducido en la versión 3.2 de iOS, es el encargado de reconocer los gestos táctiles que realicemos sobre los elementos de la interfaz gráfica de nuestra aplicación. Añadiendo un reconocedor de gestos a nuestra vista (view) indicaremos qué acciones llevar a cabo cuando un gesto (deslizar, pellizcar, toque…) ocurra. Esto ahorrará mucho código al programador.

Para reconocer los gestos utilizamos la clase *UIGestureRecognizer*, y gracias a ella podemos reconocer y definir una acción para los siguientes gestos: Toque (tap), pellizcar (Pinch, zoom), arrastrar (dragging), deslizar (swipe), rotación y toque largo.

Notifications:

Esta capa también da soporte al servicio de envío de notificaciones, ya sea locales (a partir de iOS 4) o push (a partir de iOS 3), hablaremos de cada uno de estos tipos de notificaciones:

Las *notificaciones push* proporcionan los mecanismos para alertar al usuario de la llegada de nueva información, incluso cuando la aplicación que envíe la notificación no se esté ejecutando en ese momento. Con este servicio el desarrollador puede alertar al usuario mostrando un mensaje, añadiendo una badge al icono de la app o emitiendo un sonido de alerta. Este tipo de notificaciones requiere preparar a la aplicación para que realice peticiones, y tener un servidor que las envíe.

Por otro lado tenemos las *notificaciones locales*, que funcionan de manera similar a las notificaciones push pero sin requerir la presencia de un servidor externo que envíe dichas notificaciones. Las aplicaciones que corran en segundo plano pueden utilizar este medio para requerir la atención del usuario cuando ocurra algún evento. También se pueden planificar notificaciones para una fecha concreta que se mostrarán aunque la aplicación no esté en ejecución. La principal ventaja de este tipo de notificaciones es que no requieren atención de la aplicación, sino que se programan y el sistema se encarga por sí mismo de mostrarlas.

2.1.1.2 Capa Media

Esta capa proporciona todas las tecnologías necesarias para dotar a la aplicación de audio, vídeo y gráficos. Consta de las siguientes tecnologías y frameworks que permiten incluir cualquier tipo de medio en nuestra aplicación:

- **Tecnologías:**
  - Graphics Technologies
  - Audio Technologies
  - Video Technologies
  - AirPlay
- **Frameworks:**
  - Assets Library Framework 24
  - AV Foundation Framework 25
  - Core Audio 25
  - Core Graphics Framework 26
  - Core Image Framework 26
  - Core MIDI Framework 27
  - Core Text Framework 27
  - Core Video Framework 27
  - Image I/O Framework 27
  - GLKit Framework 28
  - Media Player Framework 28
  - OpenAL Framework 29
  - OpenGL ES Framework 29
  - Quartz Core Framework

A continuación ampliaremos información sobre las tecnologías y sus frameworks más importantes:

Graphics Technologies:

Los gráficos son una parte muy importante en las aplicaciones iOS, la manera estándar de crearlos es utilizando los controles y vistas estándar que nos proporciona *UIKit* en la capa superior Cocoa Touch y dejar que el sistema los dibuje. Pero si queremos ir un paso más allá y crear nuestros propios gráficos tenemos las siguientes tecnologías a nuestra disposición para administrar los gráficos de nuestra aplicación.

1. **Core Graphics (Quartz Core Framework):** Controla los vectores 2D y el renderizado de imágenes dentro de la aplicación.
2. **Core Animations (parte de Quartz):** Proporciona opciones avanzadas para animar las vistas y su contenido.
3. **Core Image:** Proporciona opciones avanzadas para manipular vídeo e imágenes.
4. **Open GL ES and GL Kit:** Proporciona soporte para renderizado 2D y 3D usando las interfaces de aceleración hardware.
5. **Core Text:** Proporciona un sofisticado estilo y mecanismo de renderizado para el texto.
6. **Image I/O:** Proporciona interfaces para leer y escribir en los diferentes tipos de formato de imágenes.
7. **Assets Library:** Proporciona acceso a las fotos y vídeos de la biblioteca del usuario.

Si creamos nuestra aplicación teniendo en mente la pantalla Retina Display, presente en el iPhone (a partir del modelo 4) y en los nuevos iPads [7], deberemos realizar muy pocos o ningún cambio en nuestra aplicación: El sistema escala automáticamente los dibujos vectoriales para soportar la mayor resolución de pantalla; por otra parte si tenemos imágenes en nuestro contenido, *UIKit* proporciona todos los mecanismos necesarios para mostrar la imagen en mayor resolución (esta adaptación normalmente se solventará aumentando 4 veces la resolución original de la imagen).

Audio Technologies:

Las tecnologías de audio incluidas en iOS permiten reproducir audio, grabar audio a través del micrófono y activar la vibración en iPhone (no en iPad ni iPod Touch).

El sistema ofrece numerosas maneras de reproducir y capturar el audio por medio de las frameworks de la siguiente lista, ordenadas de mayor a menor nivel de interfaz de acceso. Las frameworks de alto nivel son más fáciles de utilizar aunque las de nivel inferior son más flexibles pero más complejas:

1. **Media Player Framework:** Proporciona acceso sencillo a la música del usuario, pudiendo reproducir canciones, listas…
2. **AV Foundation Framework:** Proporciona interfaces escritas en Objective-C para administrar la captura y reproducción de audio.
3. **OpenAL:** Proporciona una serie de interfaces para crear audio posicional.
4. **Core Audio Frameworks:** Simples pero efectivas interfaces para reproducir y grabar contenido. Estas interfaces se utilizan para reproducir los sonidos de alerta,

activar la vibración del dispositivo y administrar la reproducción de audio multicanal local o a través de streaming.

Las tecnologías de audio en iOS nos permiten reproducir los siguientes tipos de formatos de manera nativa: ACC, Apple Lossless (ALAC), A-LAW, IMA/ADPCM (IMA4), Linear PCM, μ-law, DVI/Intel IMA ADPCM, Microsoft GSM 6.10, AES3-2003.

Video Technologies:

Estas son las tecnologías que nos permiten reproducir archivos de vídeo en una aplicación de manera local o en streaming a través de la red. Además también permite capturar vídeo a través de la cámara en los dispositivos compatibles y usar este contenido dentro de la aplicación. A continuación explicamos las diferentes frameworks con las que puede visualizar/grabar vídeo, ordenadas de mayor a menor nivel de interfaz de acceso, recordando una vez más que la capa de nivel superior es la que más simplifica el trabajo de acceso al vídeo:

1. **Clase *UIImagePickerController:*** clase incluida en *UIKit* que nos permite grabar vídeo en los dispositivos con cámara compatible.
2. **Media Player Framework:** Conjunto de interfaces que permiten mostrar vídeos en pantalla completa o de manera parcial en nuestras aplicaciones.
3. **AV Foundation Framework:** Interfaces para administrar la captura y reproducción de vídeo.
4. **Core Media:** Incluye los tipos usados por los frameworks de nivel superior, además de proporcionar interfaces de bajo nivel para interactuar con los contenidos (media).

Las tecnologías de vídeo que incluyo iOS nos permiten reproducir vídeos en los siguientes formatos: .mov, .mp4, .m4v, .3gp; utilizando los siguientes estándares de compresión:

- H.264 hasta 1.5 Mbps a resolución 640x480 y 30 imágenes por segundo (fps), con audio AAC-LC hasta 160 kbps 48khz, en .m4v, .mp4 y .mov;
- H.264 hasta 768 kbps a resolución 320x240 y 30 imágenes por segundo (fps), con audio AAC-LC hasta 160 kbps 48khz, en .m4v, .mp4 y .mov;
- MPEG-4 hasta 2.5 Mbps a resolución 640x480 y 30 imágenes por segundo (fps), con audio AAC-LC hasta 160 kbps 48khz, en .m4v, .mp4 y .mov;

AirPlay:

AirPlay es la tecnología que permite al usuario enviar audio de manera inalámbrica desde un dispositivo con iOS a un Apple TV o a unos altavoces compatibles. AirPlay esta construido dentro del framework *AV Foundation* y la familia de frameworks del núcleo, *Core Audio*. Cualquier contenido reproducido utilizando estas librerías podrá ser fácilmente enviado a través de AirPlay ya que el sistema se encargará de ello.

A partir de iOS 5 los usuarios pueden hacer modo espejo entre el iPad y el Apple TV, esto es hacer que se reproduzca el mismo contenido en ambos dispositivos. Si el desarrollador desea que este contenido varíe en cada aparato, tiene a su disposición la clase *UIScreen* que

permite asignar una vista un objeto de dicha clase y reproducirlo en la pantalla de uno de los dispositivos. La *Media Player Framework* se ocupa también de mostrar el contenido reproduciéndose actualmente en AirPlay ("Now Playing").

2.1.1.3 Capa Core Services

La capa de servicios del núcleo contiene los servicios del sistema fundamentales que puede utilizar nuestra aplicación. Incluso si no los utilizamos directamente, muchas partes del sistema los utiliza de manera indirecta. Todos los servicios de alto nivel y frameworks que nos ofrece esta capa se muestran en la parte inferior:

- **Servicios alto nivel:**
  - iCloud Storage
  - Automatic Reference Counting
  - Block Objects
  - Data Protection
  - File-Sharing Support
  - Grand Central Dispatch
  - In-App Purchase
  - SQLite
  - XML Support
- **Frameworks:**
  - Accounts Framework
  - Address Book Framework
  - Ad Support Framework
  - CFNetwork Framework
  - Core Data Framework
  - Core Foundation Framework
  - ContentsCore Location Framework
  - Core Media Framework
  - Core Motion Framework
  - Core Telephony Framework
  - Event Kit Framework
  - Foundation Framework
  - Mobile Core Services Framework
  - Newsstand Kit Framework
  - Pass Kit Framework
  - Quick Look Framework
  - Social Framework

- Store Kit Framework
- System Configuration Framework

A continuación comentaremos los servicios y frameworks más importantes que nos ofrece esta capa de servicios del núcleo:

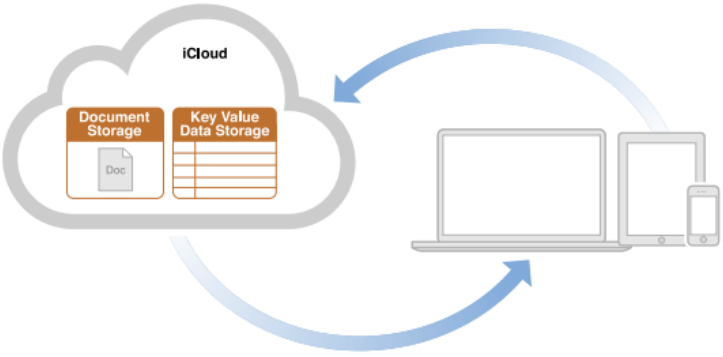
iCloud Storage:

Como hemos indicado anteriormente, el almacenamiento en la nube iCloud fue una característica incluida a partir de iOS 5 que nos permite acceder a nuestros documentos y fotos en cualquier lugar, con el único requisito de estar conectado a internet con un dispositivo de Apple. Esto quiere decir que los cambios que un usuario haga en un documento deben verse reflejados en todos los dispositivos que éste utilice sin que haya que sincronizar de manera explícita.

Existen dos maneras en las que un desarrollador puede almacenar datos de su aplicación en iCloud dependiendo del uso que se le quiera dar:

1. **iCloud document storage:** Esta herramienta se utiliza para guardar documentos en la cuenta del usuario.
2. **iCloud key-value data storage:** Esta herramienta se utiliza para guardar pequeñas cantidades de datos en diferentes instancias de aplicación en formato clave:valor.

La mayoría de aplicaciones usarán iCloud de la primera manera expuesta, guardando documentos en la cuenta de usuario para que este los edite y estén disponibles en cualquier momento y lugar. Esta es la manera en la que el usuario percibe que iCloud trabaja. La segunda manera de almacenamiento de datos en iCloud clave: valor no es una tarea percibida por el usuario. Es la manera en la que la aplicación comparte información entre diferentes instancias de la misma para restaurar una sesión, preferencias u otros fines, no siendo información imprescindible para el correcto funcionamiento de la aplicación. En la Figura 7, vemos un esquema que resume este comportamiento.



In-App Purchase:

Los desarrolladores pueden, a partir de iOS 3, ofrecer la posibilidad de que usuario haga compras dentro de una aplicación. Esto se implementa con un framework llamado *Store Kit Framework* que proporciona toda la infraestructura necesaria para dar soporte a las transacciones financieras que se realicen usando la cuenta de iTunes del usuario, dejando al programador personalizar como se muestra el contenido.

SQLite:

La librería SQLite permite al desarrollador incorporar una base de datos SQL ligera dentro de su aplicación sin tener que depender de un proceso externo ejecutándose en un servidor MySQL. Esta librería está optimizada para proporcionar un rápido acceso a la lectura y modificación de datos de las tablas.

Core Data Framework:

Esta framework está disponible a partir de iOS 3 y es una tecnología que permite administrar los datos de un modelo de controlador de vistas. Es utilizada es aplicaciones con un modelo de datos muy estructurado. En vez de definir los datos en código, se definen mediante una herramienta que proporciona un esquema gráfico de los datos del modelo de la aplicación en Xcode. Más tarde, en tiempo de ejecución, se crean instancias de estos datos que podrán ser manejados a través de este framework.

Utilizando esta herramienta se reduce considerablemente la cantidad de código que debemos escribir en nuestra aplicación además de añadirnos las siguientes funcionalidades:

1. Almacenamiento de datos en una tabla SQLite (de las estudiadas anteriormente) aumentado así el rendimiento.
2. Una clase *NSFetchedResultController* para poder introducir estos datos en *tableViews.*
3. Tratamiento avanzado de la función de entrada de texto (deshacer/rehacer).
4. Soporte para la comprobación de propiedades
5. Soporte para la propagación de cambios, garantizando que las relaciones entre objetos se mantenga consistente.
6. Soporte para la gestión de los datos en memoria.

Core Location Framework:

Esta framework proporciona los servicios de localización a las aplicaciones. Para ello utiliza la antena GPS, la red móvil o la radio Wifi, para determinar la latitud y altitud a la que se encuentra el usuario. El desarrollador puede añadir servicios de localización en su aplicación gracias a este framework, además de poder acceder a la brújula (a partir de iOS 3).

Social Framework:

Introducida en iOS 6, anteriormente se denominaba *Twitter Framework* y sólo daba acceso a la red social Twitter, esta framework pone al alcance del desarrollador una sencilla

interfaz mediante la cual puede acceder a las cuentas que el usuario tenga configuradas en redes sociales como Facebook, Twitter o Sina´s Web Service (una red de microblogs china [8]), publicando mensajes en dichas redes a través de su aplicación. Este framework funciona con el framework de cuentas para proporcionar una única acción de logueo (sig-in en inglés), garantizando así la seguridad de las cuentas.

2.1.1.4 Capa Core OS (Kernel)

Esta es la última capa de la estructura multicapa del sistema operativo iOS: la capa del núcleo del sistema o Kernel. Esta capa contiene todos los mecanismos necesarios (frameworks) para permitir que una aplicación interactúe con el hardware del teléfono. Esta capa no es usada directamente por el desarrollador de manera habitual, pero otras capas que éste sí que usa la utilizan de manera indirecta. Si queremos añadir comunicación con un hardware nuevo (drivers) deberemos de utilizar esta capa de bajo nivel. En la parte inferior pasamos a comentar todos los frameworks que nos ofrece esta capa de más bajo nivel:

- **Frameworks:**
  - Accelerate Framework
  - Core Bluetooth Framework
  - External Accessory Framework
  - Generic Security Services Framework
  - Security Framework
  - System

A continuación detallaremos los servicios que ofrecen los tres principales frameworks de esta capa del núcleo del sistema operativo:

Core Bluetooth Framework:

Esta framework permite al desarrollador interactuar específicamente con los dispositivos bluetooth conectados al dispositivo. Las interfaces de este framework, escritas en Objective-C, permiten escanear para buscar nuevos dispositivos y conectarse o desconectarse a ellos, registrar nuevos servicios bluetooth, etc.

Accelerated Framework:

Introducida a partir de iOS 4, nos proporciona las interfaces necesarias para realizar cálculos complejos como álgebra lineal o cálculos de proceso de imagen. La ventaja de utilizar este framework, en vez de crear uno por nuestra cuenta, es que este ya viene optimizado para todas las combinaciones de hardware posibles en dispositivos con el sistema operativo iOS.

External Accesories Framework:

Introducida en iOS 3, este framework proporciona soporte para la comunicación con otros accesorios hardware que conectemos a nuestro dispositivo iOS, estos pueden ser conectados a través del conector del Dock de 30 pines o a través de Bluetooth. También proporciona información sobre el dispositivo conectado y facilita la comunicación. Una vez establecida la comunicación, el desarrollador puede interactuar con dicho dispositivo ejecutando los comandos que soporte.

2.1.1.5 Conclusión

Con este apartado finaliza la explicación sobre la arquitectura multicapa de iOS, cabe destacar que en nuestra aplicación únicamente utilizamos frameworks de la primera capa, Cocoa Touch, ya que los mecanismos que nos ofrece ésta nos son suficientes para alcanzar la funcionalidad que buscamos en nuestra aplicación. Con la arquitectura multicapa del sistema operativo móvil iOS ya explicada, ya tenemos un contexto para proseguir con el análisis del otro gran pilar sobre el que se asienta el desarrollo en este sistema: el lenguaje de programación Objective-C.

2.2 El lenguaje de programación Objective-C

En este segundo apartado explicaremos las principales funciones y características de este lenguaje de programación, además de señalar las principales diferencias con respecto a otros lenguajes de programación. Después profundizaremos en Objective-C, explicando como crear objetos, la llamada y creación de métodos, herencia y demás características más avanzadas que debemos abordar para poder realizar programas de manera satisfactoria utilizando este completo pero complejo lenguaje de programación.

2.2.1 Principales características del lenguaje

Un poco de historia:

Objective –C es, al igual que C++ una extensión del lenguaje de programación C para hacerlo orientado a objetos, pero a diferencia de C++, Objective-C está basado en ideas del "mundo virtual" del lenguaje Smalltalk, [9] lo que hace de Objective-C un lenguaje más limpio que C++. Sin embargo es un lenguaje mucho menos usado que C++ excepto en el entorno de Mac OSX e iOS, donde se utiliza para programar en Cocoa y Cocoa Touch respectivamente, la API de programación de aplicaciones que ofrece Apple a los desarrolladores para estos sistemas operativos.

A diferencia de otros lenguajes de las GCC (GNU Compiler Collection [10]), Objective-C no ha sido estandarizado por ningún organismo internacional, sino fue que NeXTSTEP y ahora Mac OSX e iOS quienes han contribuido e impulsado la creación y utilización de este lenguaje. NeXT, la empresa que creo Objective-C, cedió la implementación de Objective-C a las GCC, por lo que actualmente éste forma parte de ellas [11].

Objective-C es un lenguaje que deriva de C, retrocompatible, por lo que hereda algunas características de este lenguaje como por ejemplo: sentencias de control de flujo, los tipos de datos fundamentales, estructuras y punteros, las conversiones entre tipos, el ámbito de las variables (Globales, estáticas y locales) y algunas directivas de preprocesador.

Los ficheros de código fuente tienen la extensión .m y podemos usar tanto la sintaxis de Objective-C como la de C. Además a partir de las GCC 2.95 se puede utilizar el llamado Objective-C++, que no es más que la sintaxis de Objective-C mas la de C++, estos ficheros tienen extensión .mm o .M.

Un lenguaje marcadamente dinámico:

Una de las principales características de este lenguaje es que es un lenguaje marcadamente dinámico, ya que muchas decisiones que otros lenguajes las toman en tiempo de compilación, Objective-C las toma en tiempo de ejecución. Esta característica ofrece un dinamismo que diferencia a Objective-C de otros lenguajes, permitiendo al desarrollador una depuración mucho más cómoda al poder instanciar objetos y mostrar su contenido durante ésta.

Hay cinco tipos de dinamismos que diferencian a Objective-C del resto de lenguajes que describiremos con detalle a continuación:

- **Memoria Dinámica:** Los primeros lenguajes de programación decidían cuanta memoria iban a utilizar en tiempo de compilación, pronto se dieron cuenta de este fallo e incluyeron la función *malloc()* de C que se utiliza para reservar memoria en tiempo de ejecución (dinámicamente). Posteriormente se dieron cuenta de que reservar espacio de memoria para los objetos en al pila hacía que ésta creciera bastante y podían aparecer problemas de desbordamiento de pila, por lo que lenguajes como Java u Objective-C solucionan este problema obligando a crear los objetos únicamente en memoria dinámica.
- **Tipos Dinámicos:** La tipificación puede ser de dos tipos: estática, cuando es el compilador quien lleva la cuenta de los tipos de las variables; y dinámica, cuando es el runtime [12] del lenguaje en tiempo de ejecución quien puede acceder a los tipos de las variables. Esto da lugar a la llamada introspección, característica de lenguajes como Java u Objective-C, que permite "preguntar" a los objetos en tiempo de ejecución cosas tales como: a qué clase pertenece, de qué clase deriva, qué métodos tiene, qué parámetros reciben estos métodos…
- **Carga Dinámica:** Es una característica de lenguajes como Java u Objective-C que permite en cargar solo un conjunto de clases básico al ejecutar un programa, y luego en función de la evolución del flujo de datos del programa ir cargando clases en consecuencia. Esta característica hace a los programas escritos con Cocoa extensibles (con posibilidad de instalar plugins).

Conexiones:

Las conexiones son otra cualidad que distingue a Objective-C. Interface Builder (del que hablaremos más adelante [13]) permite al desarrollador crear un grafo de objetos en un fichero .nib (NeXTSTEP Interface Builder), este fichero se cargará en memoria cuando se ejecute el programa. El fichero .nib no sólo contendrá objetos sino que también contendrá lo que se denomina conexiones, es decir relaciones entre objetos (como agregación o asociación) que se almacenan de manera persistente en el fichero. Existen tres tipos de conexiones que detallaremos a continuación:

- **Conexiones Outlet:** Son punteros de un objeto a otro objeto.
- **Conexiones target-action:** las cuales crean relaciones que permiten enviar un mensaje llamado action a un objeto llamado target. Por ejemplo cuando queremos

que se ejecute una función (action) al apretar a un botón, *UIButton*, (target) en iOS.

- **Bindings**: Permite mantener sincronizadas las propiedades de dos objetos distintos. Se utiliza principalmente en Cocoa Bindings [14]

Componentes y frameworks:

Existen dos tipos de librerías de clase de apoyo que son los llamados componentes y frameworks:

- Las librerías de componentes son un conjunto de clases que representan las distintas partes de un programa (botones, tablas, cajas de texto…). Estas librearías hacen posible la creación de conexiones entre los objetos de la interfaz.
- Por otro lado los frameworks además de disponer de las clases de los principales componentes del programa, proporcionan una serie de patrones de diseño que debe seguir el programador para construir su aplicación.

Ahora que ya hemos introducido los elementos fundamentales del lenguaje Objective C y hemos expuesto sus diferencias y similitudes con otros lenguajes de programación, vamos a pasar a profundizar sobre la sintaxis de este lenguaje en los siguientes puntos de este apartado.

2.2.2 Clases y Objetos

Clases:

Las clases en Objective-C, al igual que en otros lenguajes, constan de un fichero de interfaz (extensión .h), donde se indica la estructura de los objetos; y otro de implementación (extensión .m), que contiene la implementación de sus respectivos métodos. Una clase comienza señalando las clases a las que hace referencia y con la directiva *@interface* o *@implementation*, dependiendo si corresponde al fichero de interfaz o implantación respectivamente y finaliza con la directiva *@end.*

Objetos:

Como hemos mencionado anteriormente, los objetos en este lenguaje sólo se pueden crear en zona de memoria dinámica, esto provoca que solo nos podamos referir a ellos mediante punteros. Implicando también que solo podamos pasar a las funciones objetos por referencia, y no por valor (salvo alguna excepción).

Para instanciar un objeto debemos llamar a los métodos *alloc()* e *init()*, el primero sería en equivalente al método *new* de Java o C++, y sirve para reservar un espacio en memoria dinámica; mientras que el segundo se correspondería con el constructor en dichos lenguajes, y se encarga de inicializar las variables. A continuación veremos como usar estos métodos de instancia

```
NSString *objetoString = [NSString alloc];
objetoString = [objetoString init];
```

Es muy común llamar a estas dos funciones juntas, por lo que también podemos llamarlas en una sola línea de la siguiente forma:

```
NSString *objetoString = [[NSString alloc] init];
```

Debido a que los objetos se almacenan en memoria dinámica, es importante liberar ésta una vez que no vamos a utilizarlos. Para ello debemos de desactivar el apartado "Objective C Automatic reference Counting" dentro de la propiedades de nuestro proyecto (en Xcode) para seguidamente poder llamar al método *release* de la clase *NSObject,* que se encarga de disminuir en una unidad el número de referencias a dicho objeto, liberando espacio en memoria cuando no quede ninguna:

```
[objetoString release];
```

Objetos estáticos:

Ya hemos comentado anteriormente que otros lenguajes de programación como Java o C++ solo pueden referirse a sus objetos de manera estática, esto es que el compilador conoce el tipo de estos objetos, o el tipo de los punteros objetos. Por lo que a los punteros que hagan referencia a un tipo se les denominan **punteros a objetos estáticos.**

Objetos dinámicos:

Este tipo de objetos es una característica que diferencia a Objective-C del resto de lenguajes. Permite que el tipo de los objetos sea conocido solamente por el runtime y no por el compilador. A este tipo de objetos se les denomina **punteros a objetos dinámicos** y se declaran como variables de tipo *id*, veremos un ejemplo a continuación:

```
id objetoString = [NSString new];
```

Se puede observar que el tipo *id* no lleva el asterisco característico de un puntero, esto es así porque *id* es un puntero en sí mismo. Al llamar a los métodos y funciones de un objeto del tipo *id* el compilador no comprueba que existan dichos métodos en tiempo de compilación, sino en tiempo de ejecución, generando un "runtime error" (error en tiempo de ejecución) si

estos no se encuentran implementados en la respectiva clase del objeto. Objective-C produce avisos de *warning* en tiempo de compilación sino se encuentran dichos métodos, aunque el programa supera con éxito esta fase de y se ejecuta.

Identificadores NULL, Nil y nil:

Además del identificado de preprocesador *NULL* (heredado de C), Objective-C también posee otras palabras reservadas para hacer referencia a objetos sin inicializar en el caso de *nil,* y a clases o metaclases en el caso de *Nil*. Todos estos objetos hacen referencia al valor 0, por lo que son intercambiables, pero este significado merece la pena recordar.

Variables de instancia:

En caso de querer acceder a las variables de instancia de un objeto desde fuera de su propia clase se utiliza, al igual que en C++, el operador flecha (->). Este operador solo se puede utilizar con objetos estáticos (no con objetos tipo *id*). Dentro de un método de la misma clase se puede acceder directamente a las variables de instancia bien indicando su nombre, o usando el identificador *self*, que se corresponde al identificador *this* de Java o C++.

Métodos de objetos:

Los métodos son operaciones asociadas a un objeto. A diferencia de C++ o Java, en Objective-C el método a ejecutar siempre se decide de manera dinámica, por lo que es más correcto hablar de *enviar un mensaje a un objeto*, en vez utilizar la expresión *llamar a un método* en Objective-C, aunque en la práctica estos dos términos se utilizan de manera ambigua. Cabe destacar también que se pueden declarar métodos privados que solo podrán ser accesibles desde dentro de la misma clase.

A continuación explicaremos como se realiza la declaración de métodos en Objective-C y cómo debemos enviar mensajes a los objetos para que ejecuten estos métodos. Empezaremos comentando cuáles son las partes que componen la declaración de un método:

- **Nivel:** es obligatorio, e indica quien recibe el mensaje. Si es la clase (en cuyo caso se pone el símbolo + al principio de la declaración) o la instancia de un objeto (en cuyo caso se pone el símbolo -). Sería similar al identificador *static* de Java.
- **Tipo de retorno:** Indica el tipo de variable que devuelve el método. Puede ser *void* si no devuelve nada; por defecto, si no se especifica tipo de retorno, se devuelve un objeto id, que correspondería con un *int* (al igual que en el lenguaje C).
- **Nombre del método:** es obligatorio, y sirve, junto con el nombre de los parámetros, para identificar al método de manera única en la clase del objeto.
- **Parámetros:** Que utilizan una notación heredada de Smalltalk diferente a la vista en otros lenguajes. Permite identificar por medio de un identificador los diferentes parámetro que recibe dicho método, tomando estos identificadores el nombre de etiquetas. Cuando el método recibe un solo parámetro se toma como etiqueta del éste el propio nombre del método. Al igual que las funciones C, los métodos en Objective-c permiten ser llamados con un número variable de parámetros.

Veremos una muestra de este esquema de declaración de métodos en el ejemplo que exponemos a continuación:

```
- (void) actualizarCoordenadasX: (int) coord_x Y: (int) coord_y;
```

Este método tiene nivel de objeto (-), no tiene tipo de retorno (void) y recibe como argumentos dos coordenadas X e Y, la coordenada X es identificada por quién llama al método por el nombre del método y la Y por la etiqueta 'Y'. Mientras que estos parámetros son reconocidos dentro del método con los identificadores *coord_x* y *coord_y* respectivamente. La declaración equivalente a este método en Java correspondería al siguiente fragmento de código:

```
public void actualizarCoordenadasXY(int coord_x,int coord_y);
```

Para realizar la invocación del método, es decir, el envío de mensaje al objeto (*satelite*), utilizamos la siguiente notación:

```
[satelite actualizarCoordenadasX: nueva_coordenada_x Y: nueva_coordenada_y]
```

Lo que en lenguaje de programación Java correspondería al siguiente fragmento de código:

```
satelite.actualizarCoordenadasXY(coord_x,coord_y);
```

Podemos observar que esta notación puede resultar un poco liosa comparada con la sintaxis de Java o C en un principio, pero una vez adquiramos cierta soltura, será una cualidad que echaremos en falta en estos lenguajes.

Encapsulación:

La encapsulación nos permite ocultar partes del código de nuestros objetos que otros programadores no necesitan conocer para utilizarlos. Esto se denomina el ámbito de instancia de las variables de un objeto y para definirlo en Objective C, se utilizan los llamados modificadores de acceso @private, @protected y @public:

- **@public:** Una variable con este modificador de acceso será accesible desde cualquier parte del programa.
- **@private:** Una variable con este modificador de acceso sólo será accesible desde el mismo Objeto (clase)
- **@protected:** Una variable con ese modificador de acceso actuará de una manera similar a @private, salvo porque también será accesible desde las una clase derivada de ésta.

Una peculiaridad de Objective-C que no encontramos en otros lenguajes como Java o C++, es que los modificadores de acceso no afectan a los métodos, pudiendo ser estos de dos tipos: públicos si están definidos dentro del fichero de interfaz, y privados si se encuentran dentro del fichero de implementación. Además el compilador nos permite invocar métodos privados sin generar error de ejecución, sino generando un *warning* en tiempo de compilación, este comportamiento no es definitivo y podría variar en un futuro, tal y como avisa dicho *warning*.

Los objetos creados sin clase base, es decir, no heredan ninguna clase, son utilizados normalmente en Objective-C como estructuras de datos personalizadas, tal y como muestra el siguiente ejemplo donde se crea un "objeto fecha":

· Debido a que la clase no requiere métodos, no es necesario crear un fichero de implementación.

```
@interface Fecha {
    @public
    NSInteger dia;
    NSInteger mes;
    NSInteger anyo;
}
@end
```

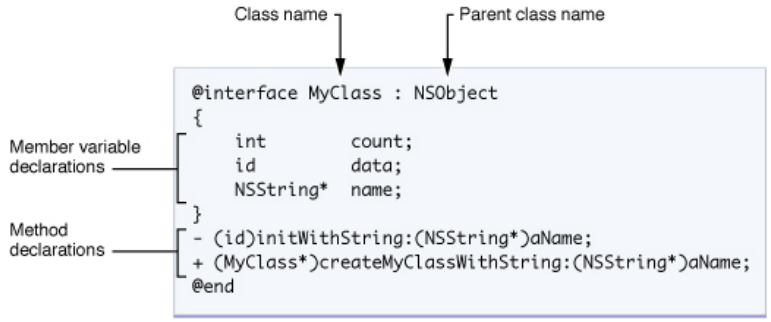
Con todo esta base explicada ya tenemos una noción de los conceptos más básicos de los que consta este lenguaje, a continuación, en los siguientes apartados de este tema, analizamos aspectos más avanzados del lenguaje de programación Objective-C tales como: la herencia y receptores, categorías, protocolos y delegados, los objetos de colección que nos ofrece este lenguaje, o el ciclo de vida de un objeto y la gestión de memoria.

Figura 8. Representación de herencia en lenguaje Objective-C

2.2.3 Herencia y receptores

Herencia:

La herencia consiste en la creación una clase llamada clase derivada cuya creación se basa en otra clase, llamada clase base, de la que heredará (podrá acceder) sus métodos y variables de instancia. La clase derivada puede sobrescribir métodos de la clase raíz (la clase base), pero al contrario que lo que ocurre en C++, en Objective-C no se pueden volver a definir las variables de instancia de la clase base en la clase derivada. La siguiente ilustración muestra como se plasma este comportamiento en el código de Objective C, donde *MyClass* es la clase derivada y *NSObject* (de la que derivan todos los objetos en Objective-C) la clase raíz, también aparecen los métodos y atributos de dicha clase derivada:



Receptores:

Los receptores son objetos a los que se envían mensajes, hay dos de ellos que son muy importantes y juegan un papel decisivo en el ámbito de la herencia: *self* y *super*; de los cuales hablaremos a continuación:

- **Self:** No es más que un puntero apuntando a la clase a la que se ejecuta el método, equivale al *"this"* de Java o C++. Solo tiene sentido utilizarlo dentro de un método. Este receptor se comprueba en tiempo de ejecución y se puede modificar, al contrario que *this* de C++ o Java que es de solo lectura.
- **Super:** Proporciona el mecanismo para saltarse los métodos de la clase e ir a la clase base a la hora de ejecutar un método, es decir, impide que se busque el método en la clase en la que nos encontramos. Este receptor se comprueba en tiempo de compilación y solo se pude leer. Con super podemos invocar métodos de la clase raíz de manera directa, tal y como se muestra en el siguiente ejemplo, en el que se invoca al método *viewDidLoad* de la clase raíz, *UIViewController* [15]:

```
[super viewDidLoad];
```

2.2.4 Categorías, protocolos y delegados.

Las clases son la manera más importante de añadir métodos (funcionalidades) a un objeto, pero no la única. Podemos añadir métodos a una clase de tres diferentes maneras más: categorías, protocolos y extensiones. No centraremos en las dos primeras.

Categorías:

Las categorías nos permiten modificar una clase ya existente aunque no dispongamos de su código fuente. La diferencia que hay entre la herencia y la categorización es que la herencia sólo nos permite crear nuevas hojas en la jerarquía de clases, mientras que la categorización nos permite modificar nodos interiores de la jerarquía de clases.

La categorías tienen varias finalidades, como por ejemplo: utilizarlas para implementar una clase en distintos ficheros de código fuente, esto nos permite dividir la clase en métodos lo que puede ayudar a dividir el trabajo de implementación entre grupos de programadores. También permiten añadir a las clases librearías de otros fabricantes, pudiendo así añadir funcionalidades que nos hubiese gustado que hubiese incluido la clase original; también se pueden usar para distinguir entre la API pública y la API dirigida al framework de nuestra clase; y finalmente, como veremos más adelante, las categorías ayudan a la declaración de protocolos informales.

La declaración de la interfaz una categoría es igual a la de una clase salvo que se pone el nombre de la categoría ente paréntesis después del nombre de la clase a la que añade la categoría. Apple recomienda renombrar los ficheros de la categoría con el formato: *nombre_clase+nombre_categoría.m* y *nombre_clase+nombre_categoría.h,* respectivamente.

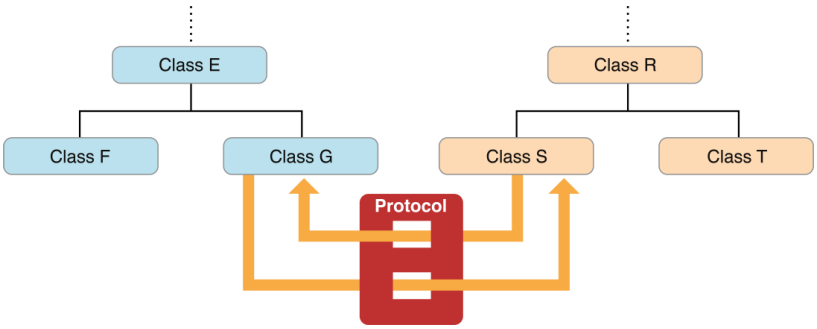
Protocolos:

Las categorías declaran métodos asociados con una clase en particular, mientras que los protocolos, por otra parte, declaran métodos no asociados a ninguna clase en particular, sino que cualquier clase o conjunto de clases los puede implementar. Según esto se vemos que un conjunto de clases no relacionadas entre sí en un principio pueden acabar compartiendo un conjunto de métodos a través de un protocolo. Un ejemplo a continuación, este protocolo desencadenará un evento (invocación de método) cuando se haga click con el ratón (en Mac OSX):

```
- (void)mouseDown:(NSEvent *)theEvent
```

Se dice que una clase adopta un protocolo si dispone de los métodos que reciben los mensajes de dichos protocolos, de esta manera podremos dividir las clases por el hecho de compartir una funcionalidad común, y, usando las categorías, podremos dividir las clases también por jerarquía.

Figura 9. Esquema de funcionamiento de un protocolo



A continuación detallaremos los 6 principales escenarios donde los protocolos pueden resultar de utilidad:

1. Para declarar métodos que otros deben implementar.
2. Para conseguir similitud entre clases que no están relacionadas, para que compartan así un conjunto de operaciones.
3. Para declarar la interfaz de un objeto ocultando su clase, pudiendo así crear un *objeto anónimo* para el resto de programadores.
4. Para indicar a varios fabricantes la interfaz común que debe de tener un objeto. Y poder soportar dispositivos de varias compañías el software creado.
5. Para reducir la complejidad de un objeto, dando a conocer menos métodos de los que en realidad tiene la interfaz clase.
6. Para hacer una comprobación estática de tipos. Pudiendo así por ejemplo provocar que en tiempo de ejecución el compilador compruebe si un objeto tiene los métodos que queremos ejecutar.

Para hacer que una clase adopte un protocolo debemos de poner el nombre de los protocolos que adopte entre paréntesis angulares dentro del fichero de interfaz (.h), separando por comas los nombres de los diferentes protocolos. A continuación mostramos un ejemplo:

```
@interface MyController : UIViewController <UITableViewDelegate>
{
}
```

En este ejemplo podemos ver que la clase creada *MyController* (que deriva de la clase UIViewController) adopta el protocolo **<UITableViewDelegate>** [16]. Cabe resaltar la importancia de este protocolo ya que es el que ofrece acciones tan esenciales en aplicaciones iOS como por ejemplo: indicar cuantas filas se van a mostrar en la pantalla, el número de secciones, o el contenido (celdas) de dichas filas. Es usado en nuestra aplicación en muchas de sus vistas (views) entre otros protocolos.

Para finalizar cabe destacar, que existen dos tipos de protocolos: **formales** e **informales**, el primero obliga a la clase a implementar todos sus métodos, comprobándose esto en tiempo de compilación; por otra parte los informales son protocolo son el que cuyos métodos podrían opcionalmente implementar otras clases. Los delegados, que comentaremos a continuación, se basan en protocolos informales.

Delegados:

Hay diseños de software en los que resulta interesante que un objeto pueda delegar parte de su trabajo a otro objeto llamado objeto delegado. Cocoa suele recomendar la utilización de delegados en lugar de utilizar la derivación de sus clases de las librearías, por lo que son utilizados con mucha frecuencia por los desarrolladores.

Dado que el delegado no tiene porqué implementar todos los métodos, la manera de implementar el delegado es mediante el uso de un protocolo informal.

2.2.5 Objetos de colección

Los objetos de colección que nos ofrece el lenguaje de programación Objective-C están pensados para agregar punteros a objetos, realizando automáticamente las llamadas *retain* y *release*, relacionadas, como hemos indicado anteriormente, con la gestión de memoria. Expandiremos información en este punto sobre los mecanismos de colección que más hemos usado en nuestra aplicación: *Arrays* y *Diccionarios.*

Arrays:

Un array es un objeto básico de colección de datos, almacena objetos *indexados*, éstas pueden ser de tres tipos:

- **NSArray:** Representa un objeto array inmutable, es decir, no se puede modificar su contenido tras su creación, pero si podemos modificar en contenido de los objetos que almacena.
- **NSMutableArray:** Es igual que un *NSArray* salvo porque, como su propio nombre indica, es mutable, es decir, se puede añadir objetos y borrarlos del array una vez creada. Si el número de elementos no va a variar, es más eficiente utilizar un *NSArray.*
- **NSPointerArray:** Es una versión de array más flexible que las dos anteriores.

En la documentación de dichas clases podemos encontrar información mas extendida sobre los métodos que soportan, existen métodos para iniciar Arrays con objetos, ficheros, datos… Métodos para saber cuantos objetos contiene un Array, o para saber si contiene un elemento en concreto. Documentación de clases: *NSArray* [17], *NSMutableArray* [18] y *NSPointerArray* [19].

Diccionarios:

Un diccionario es una colección de *pares objeto:valor*, a cada uno de estos pares de le denomina entrada del diccionario. En Objective-C se pueden representar los diccionarios utilizando tres clases diferentes:

- **NSDictionary:** Representa un diccionario que una vez creado no podremos modificar sus valores ni sus claves asociadas, no podremos añadir o borrar entradas, no podremos modificar el objeto que actúa como *clave*, pero si el que actúa como *valor.*
- **NSMutableDictionary:** Es un subclase (clase derivada) de NSDictionary, añade la funcionalidad de añadir o borrar entradas.
- **NSMapTable:** Implementa un diccionario más avanzado y flexible que los dos anteriores.

En la documentación de dichas clases podemos encontrar información mas extendida sobre los métodos que soportan, existen métodos para iniciar un diccionario utilizando los elementos de otro (pudiendo elegir entre copiar sus elementos en nuevas posiciones de memoria o utilizando las direcciones de los objetos del diccionario original, lo que provocaría que el cambio se refleje en ambos diccionarios al modificar uno de ellos); saber cuantos elementos contiene el Diccionario, si contiene una determinada clave o elemento, podemos generar un Array con todas las claves de dicho diccionario… Añadimos la documentación de dichas clases a continuación para extender esta información: *NSDictionary* [20], *NSMutableDictionary* [21] y *NSMapTable* [22]

2.2.6 Ciclo de vida de un objeto y gestión de memoria

Ciclo de vida de un objeto:

Todas las clases de Objective-C deben implementar métodos que manejen el ciclo de vida de un objeto: Creación, inicialización, copia y destrucción. La clase *NSObject* [23] proporciona métodos para ello (**initialize, alloc, dealloc, init, new** y **copy**), y como todos los objetos derivan de esta clase, todos ellos podrán utilizarlos. Analizaremos estas fases y los métodos más importantes que se deben usar en cada una de ellas:

- **Fase de creación e inicialización:** Como hemos comentado anteriormente, la creación del objeto se realiza utilizando el método **alloc(),** que reserva el espacio en memoria necesario para al almacenar el objeto; mientras la inicialización se realiza utilizando el método **init()**. Una vez hemos realizado estas dos acciones, podremos utilizar el objeto.
- **Fase de desinicialización y liberación:** En C++ esta función se realiza usando el método *delete***,** mientras que en Java es el recolector de basura el que se encarga

de ello. La clase NSObject ofrece el método ***dealloc()*** para este fin, este método se encarga de liberar el espacio en memoria de los objetos y no tiene valor de retorno (devuelve *void*). Es tarea del programador llamar al método ***release(),*** que elimina una referencia al objeto. Cuando no queden referencias al objeto se llama al método ***dealloc()*** que se encarga de eliminar dicho objeto de memoria.

- **Copia de un objeto:** Como hemos comentado, en Objective-C todos los objetos se definen como punteros, por ello es posible crear un objeto a partir de una copia de otro original, siendo interpretados como dos objetos independientes dentro del sistema (es decir utilizando dos posiciones de memoria diferentes). Esto se realiza utilizando el método **copy()**

Gestión de memoria:

En C++ la reserva y liberación de memoria dinámica es responsabilidad única y exclusiva del programador, en Java se utiliza el llamado recolector de basura, mientras que Objective-C se encuentra entre estas dos aproximaciones ala hora de la liberación de memoria: El programador puede hacerlo manualmente él solo, usar el runtime para que le ayude en dicha tarea, u olvidarse por completo y delegar esta tarea al recolector de basura.

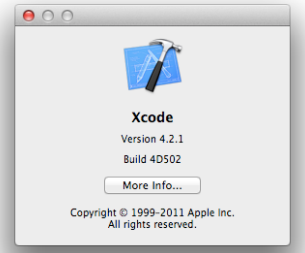
La clase *NSObject* de Cocoa usa una técnica para la liberación de memoria que tiene en cuenta las referencias actualmente existentes a un determinado objeto para liberar su espacio en memoria cuando éstas lleguen a cero. A diferencia de la técnica de recolector de basura, ésta requiere la cooperación del programador para decrementar el número de referencias cada vez que no se vaya a volver a usar un objeto. Por lo que es necesario que el programador se acostumbre a seguir una serie de patrones de programación, que consisten principalmente en utilizar los siguientes métodos:

- Cuando se crear un objeto, con el método ***alloc()*** o ***copy()***, automáticamente se inicializa a 1 la cuenta de referencias a ese objeto, cuando un objeto recibe el método ***retain(),*** este número de referencias aumenta a una unidad. Se deben usar estos métodos cuando queramos indicar que usamos un objeto.
- El método ***release()***, se utiliza para decrementar el número de referencias a ese objeto, tras llamar a este método se comprueba que si el número de referencias es cero, y de ser así, se llama al método ***dealloc()*** para liberar el espacio en memoria que estaba ocupando dicho objeto.

La regla general para liberar la memoria que ocupa un objeto es que éste haya recibo un número de mensajes **release** mayor en una unidad al número de mensajes **retain** que haya recibido. Es importante recordar esta normal a la hora de desarrollar para un dispositivo con el sistema operativo iOS, ya que la cantidad de memoria RAM de estos dispositivos es limitada y es un requisito imprescindible controlar los recursos que consumen nuestros objetos.

2.3 Entorno de desarrollo Xcode

Con este resumen finalizamos nuestra introducción al lenguaje de programación Objective-C, habiendo adquirido una base suficiente como para ponernos a programar, bien sea para el sistema operativo de escritorio Mac OSX o para el sistema móvil iOS. Se utilizará para ello el entorno de desarrollo **Xcode,** suministrado de manera gratuita (la versión 4.0 costaba 3.99€, pero a partir de la 4.1 volvió a ser un software gratuito) por Apple para tal fin. Analizaremos este entorno de desarrollo con detalle en este tercer apartado del capítulo:



Xcode es un entorno de desarrollo creado por Apple Inc. [24] y se suministra gratuitamente para Mac OSX a través de la Mac App Store [25]. La versión que se ha utilizado para desarrollar la aplicación que se analiza en esta memoria ha sido a la versión 4.2.1, como se puede ver en la imagen de la izquierda.

También incluye la colección de compiladores GNU (GCC), por lo que puede compilar código de C, C++, Objective-C, Java y Apple Script, mediante una amplia gama de modelos de programación. Pasaremos a analizar las principales características de este completo y funcional entorno de desarrollo.

2.3.1 Principales características

En primer lugar se mostrarán unas capturas explicando sus principales funciones, tales como la creación de un proyecto o descripción del espacio de trabajo y principales herramientas:

Creación de un proyecto e interfaz gráfica de Xcode (UI):

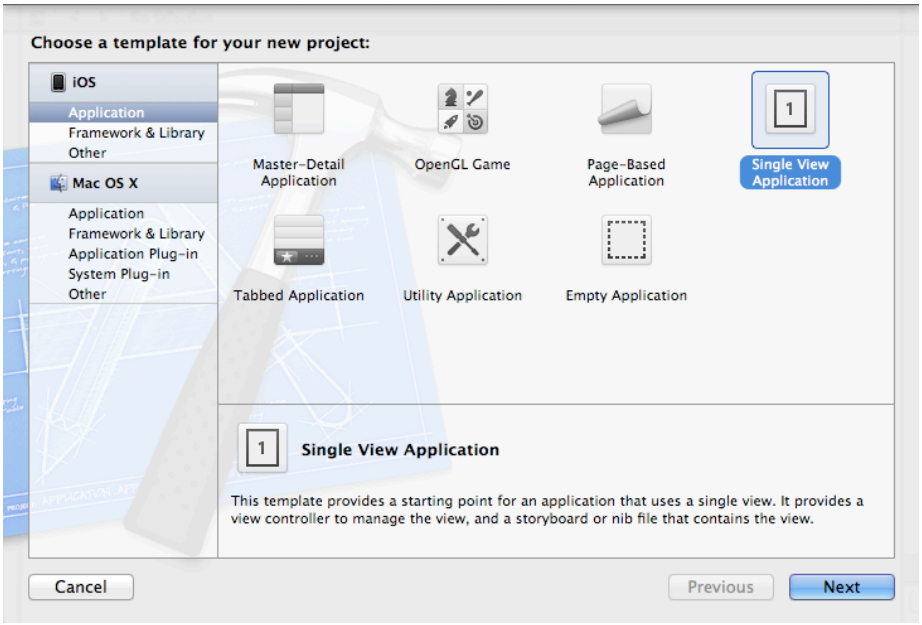


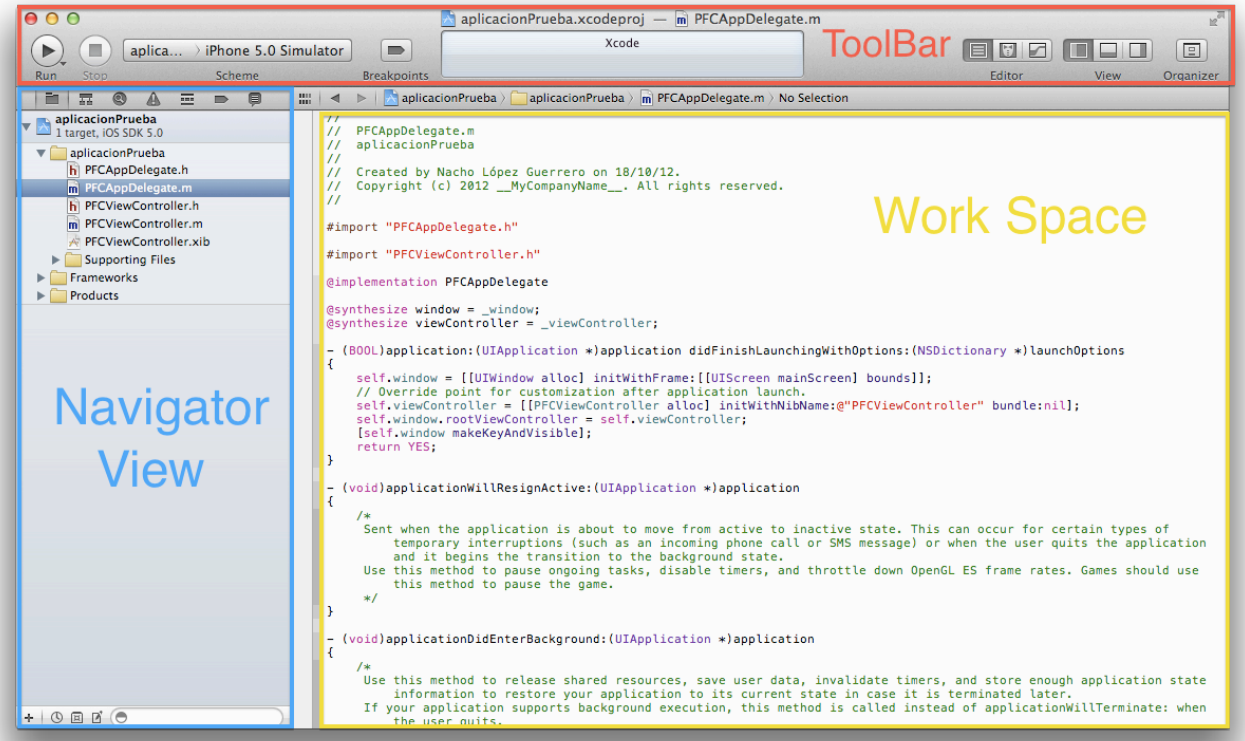
Figura 10. Creación de nuevo proyecto en Xcode

En la Figura 10 de la parte superior vemos una captura de pantalla de la ventana que se encuentra el usuario al abrir dicho programa y crear un nuevo proyecto. Aquí se muestra unas selección de diferentes plantillas para realizar nuestra aplicación, la información de cada plantilla se puede leer en la parte inferior de la ventana: por ejemplo se creará una vista con su respectivo controlador si seleccionamos la opción "*Single View Application*"; o la plantilla necesaria para crear una aplicación maestro detalle seleccionando "*Master-Detail Application*", o una plantilla vacía para poder crear así nosotros la interfaz directamente a partir de código, si así lo deseamos.

A continuación vamos a mostrar un ejemplo de lo que ocurriría si el usuario seleccionase la opción Single View Application, que proporciona la plantilla necesaria de un archivo de vista *ViewController.xib* y respectivos ficheros de clase del controlador de vista, *ViewController.h* y *ViewController.m*. El archivo .xib se utiliza para personalizar la interfaz gráfica con una herramienta incluida en XCode llamada Interface Builder, del que hablaremos más adelante.

Tras crear un proyecto, aparecerá la pantalla principal del entorno de trabajo que se muestra más abajo destacando sus tres principales zonas: la barra de tareas (ToolBar), el Navegador de Vistas (Navigator View) y el espacio de trabajo (Work Space), de las que hablaremos a continuación.

Figura 11. Interfaz gráfica del entorno de trabajo Xcode



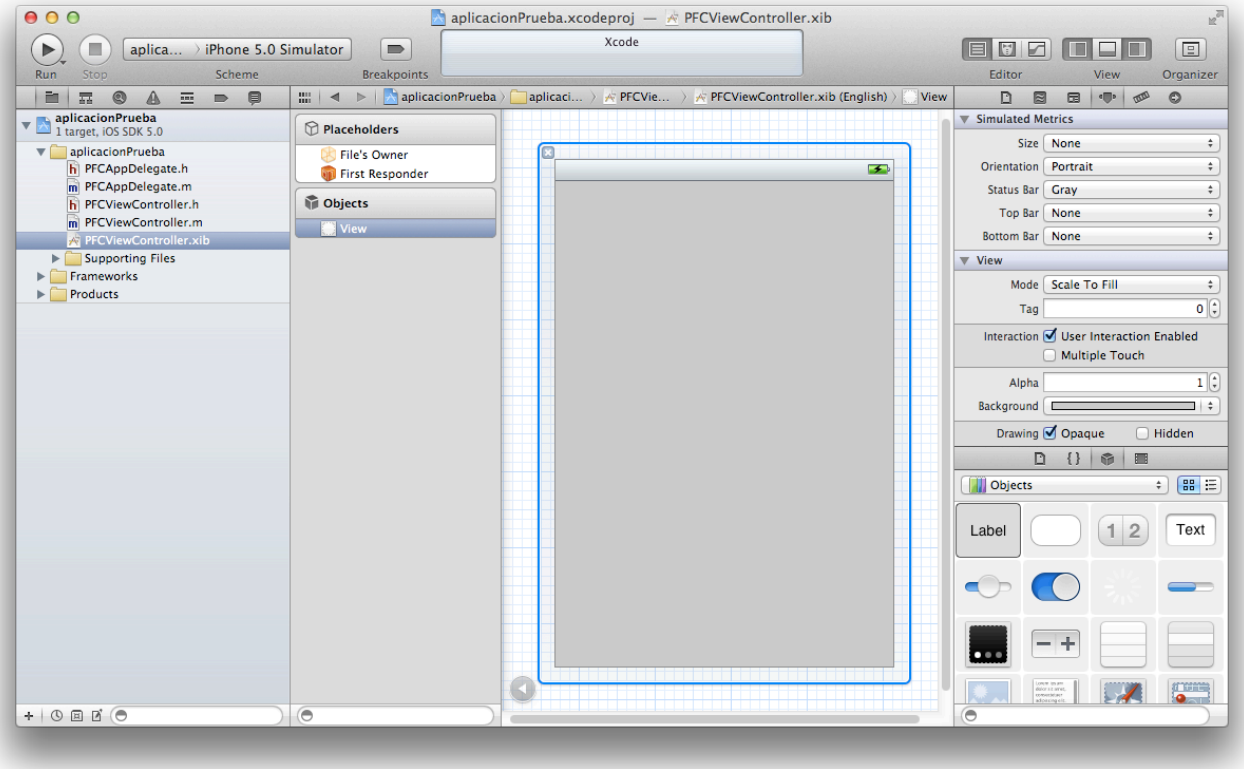
- **Navigator View:** Es el navegador de clases de objetos (.h .m), vistas (ficheros .xib) de nuestro proyecto y todos los recursos que éste necesita.
- **Tool Bar:** Es la barra de tareas donde encontramos botones con los que podemos iniciar/detener la ejecución de la App que se esté desarrollando. También encontramos una caja en mitad de la barra que proporciona información sobre el proyecto (como el número de errores por ejemplo), conocida como "*Activity View*". En la parte derecha podemos observar 2 cajetillas con 3 botones cada una: la primera de ellas se denomina *Editor* y nos deja elegir entre tres maneras diferentes de mostrar la información en el *Work Space*: la vista estándar (por defecto), la vista de asistente, que muestra una vista partida en dos del proyecto, y es especialmente útil para conectar Outlets de la interfaz grafica .xib con partes del código del controlador (.m); y por último la vista de versiones, en la que podemos cambiar entre diferentes versiones anteriores de nuestro proyecto. La segunda cajetilla de tres botones muestra y esconde elementos de la interfaz.
- **Work Space:** Es la vista principal de Xcode. En ella se muestra el contenido del fichero de nuestro proyecto que hayamos seleccionado en el Navegador de Vistas (Navigator View). En la parte superior podemos observar la llamada **Jump Bar** que nos permite navegar por nuestro proyecto de manera iterativa; y en la parte izquierda encontramos una zona de columna donde poder incluir *Break-Points* para facilitar la depuración.

Vemos que dentro de nuestro proyecto, aparte de los conocidos ficheros de clases, encontramos la clase **AppDelegate** esta clase es la encargada de dar la información necesaria sobre la ejecución de la aplicación tal como la primera vista que se cargará al iniciarse. También observamos un fichero de interfaz gráfica con extensión .xib (llamado fichero nib - NeXTSTEP Interface Builder -), **ViewController.xib,** al hacer click sobre este fichero, la herramienta que se presente en el espacio de trabajo pasará de ser el clásico editor de textos, a una herramienta conocida como **Interface Builder,** que analizaremos en el siguiente punto.

Figura 12. Interfaz gráfica de la herramienta Interface Builder

2.3.3 Interface Builder

Interface Builder es la herramienta de diseño de interfaz gráfica que viene incluida con Xcode. Permite construir y personalizar la interfaz que tendrá la aplicación. Esta herramienta se mostrará en el *Work Space* automáticamente al seleccionar un fichero nib con extensión .xib, y si se hace doble click sobre dicho fichero se mostrará en una nueva ventana. A continuación aparece una captura de esta herramienta:



Podemos particionar la interfaz de Interface Builder en tres grandes campos:

- El de más a la izquierda dónde se encuentran los aparatados **Objects** y **Placeholders: Objects** agrupa los objetos que componen nuestra interfaz, tales como vistas (view), pickers, botones, tab bars… Por otra parte podemos observar que el contenerdor **Placeholders** tiene dos objetos creados por defecto, **File´s Owner** que hace referencia al objeto que ha cargado el fichero nib del disco, es decir, hace referencia a la clase propietaria de dicho nib que contendrá los métodos (acciones) que se ejecutan al darse algún evento en la interfaz representada por dicho fichero nib; y **First Responder,** que

representa el objeto de la interfaz con el que el usuario está interactuando en un momento determinado. Por ejemplo si el usuario está introduciendo texto en un *textField,* dicho *textField* tomará en ese momento la identidad de First Responder.

- El segundo campo correspondería al campo de trabajo y es donde situaremos los elementos (objetos) que compondrán nuestra interfaz y nos permite hacernos una idea gráfica de como será el aspecto estético de nuestra aplicación.

- Por último, en la parte de más a la derecha encontramos dos contenedores: el contenedor superior que varía dependiendo del icono que hayamos seleccionado entre los seis que tiene en su parte superior (File inspector, Quick help inspector, Identity inspector, Attributes inspector, Size inspector y Connections inspector), dejándonos configurar aspectos de los objetos que componen la interfaz tales como sus atributos o las conexiones que tienen con los objetos y métodos definidos en la clase seleccionada como **File´s Owner.** Por otra parte, en el contendor inferior, la llamada **Librería,** es donde podemos seleccionar los objetos que formarán parte de nuestra interfaz entre todos los que componen la librería de Interface Builder.

Merece la pena indicar también que no es imprescindible el uso de Interface Builder para la realización de la interfaz gráfica de la aplicación. Es decir, sin utilizar ningún fichero nib, usando solamente código a partir de los delegados que nos ofrecen los diferentes objetos de interfaz de iOS de Xcode, podemos crear una completa interfaz gráfica que utilice objetos y métodos de la librería que nos ofrece Xcode.

2.3.4 iPhone Simulator

El **iPhone Simulator** es un software que nos ofrece Apple para poder probar nuestras aplicaciones sin necesidad de instalarlas en el dispositivo que tengamos en asociado en 'modo desarrollo', o en caso de que no dispongamos de una cuenta de desarrollador para ello. Este programa, que se lanza automáticamente al compilar una app para iOS, emula el comportamiento del teléfono y nos permite hacer los mismos gestos que podríamos hacer con un iPhone real, incluyendo el uso del botón Home. Nos muestra como se vería nuestra aplicación en un terminal físico. Se instala conjunta y automáticamente con Xcode.

3 Estructura multicapa de la Aplicación

3.1 Descripción global de la aplicación

La utilidad final de nuestra aplicación es, como ya hemos comentado brevemente en el capítulo 1, permitir al usuario la personalización del tipo y nivel de molestia de las notificaciones que le envían las diferentes aplicaciones (servicios) instalados en el teléfono. Supliendo así las diferentes necesidades de alerta que tenga cada uno de los diferentes tipos de usuarios. Gracias a ello se permite ampliar el nivel de configuración que ofrece por defecto el sistema operativo en el ámbito del envío de estas notificaciones. Para lograr este fin, la aplicación permite definir una serie de perfiles con un nivel diferente de intromisión cada uno, pudiendo configurar, para cada perfil, el nivel de molestia de los diferentes mecanismos de notificación que proporciona el sistema para alertar al usuario. El usuario podrá asociar uno de estos perfiles a una aplicación determinada, controlando así el grado de intromisión de las notificaciones que enviará dicha aplicación según sus propias necesidades, que pueden ser muy dispares dependiendo del tipo de persona que utilice el dispositivo y/o de la aplicación que envíe dichas notificaciones.

El usuario podrá además configurar el comportamiento de estas notificaciones de manera avanzada pudiendo crear **transiciones** entre estos perfiles asociados a las aplicaciones, cambiando así el grado de alerta de las notificaciones de una determinada aplicación según el contexto en que se encuentre el usuario. Para ello, para todos o parte de los servicios que tengan asociados el mismo perfil, se podrá intercambiar dicho perfil asociado por otro perfil existente cuando se cumplan una serie de condiciones. El usuario podrá configurar a su total necesidad qué servicios de los asociados al perfil (en adelante **Perfil From**) se verán afectados por la transición y qué condiciones la provocaran, así como crear nuevas condiciones o editar la estructura interna de las ya existentes.

En definitiva, nuestra aplicación provee toda la infraestructura necesaria para la realización de todas estas tareas. Analizaremos durante este capítulo la arquitectura que hemos seguido para desarrollar la aplicación, la **arquitectura multicapa**, pero antes de ello explicaremos de manera detallada cuales son todos los elementos que intervienen en las tareas que realiza nuestra aplicación descritas anteriormente, logrando así dar una mejor idea de la funciones que desempeñan estas tareas y los datos que manejan para llevarlas a cabo.

3.1.1 Datos que utiliza la aplicación

En primer lugar explicaremos los datos de los que se compone nuestra aplicación, y cuál será su interpretación y utilidad dentro de la aplicación:

Perfiles:

Son los perfiles que permiten la personalización del servicio de notificación. Para ello, cada perfil almacena un nivel diferente de intromisión para cada uno de los mecanismos de alerta que nos ofrece el teléfono. Los mecanismos de interacción considerados en este proyecto son: sonido, vibración, señal led cámara, texto en barra de estado (status bar), toast (mensaje en parte inferior de pantalla), speech (servicio accesibilidad de voz) y dialog (mensaje mostrado en pantalla). Cada perfil almacenará, además de su nombre, un valor que variará entre **1, 2 y 3** para cada uno de estos mecanismos de notificación, dicho valor indicará el nivel de intrusión del mecanismo al que se asocie. Por ejemplo el valor **1** está asociado al nivel de molesta más bajo donde el usuario no se molesta para nada; el valor **2** se asocia a un nivel de molestia medio donde el usuario tiene que hacer un esfuerzo para ser consciente de la notificación; y el valor **3** se asocia al máximo nivel de intrusión donde el usuario es totalmente consciente de las notificaciones recibidas. Los mecanismos de interacción asociados a cada nivel de intromisión poseen ciertas restricciones entre sí que indicaremos a continuación:

- **Speech** y **Sound** no pueden estar activados al mismo tiempo.
- Los mecanismos **Toast, Status Bar** o **Dialog,** no pueden estar activos al mismo tiempo.

Mostramos una tabla con cada uno de los posibles valores que puede tomar cada mecanismo de interacción con el usuario, y a qué estado de alerta representa:

	Vibration	Sound	Light Signal	Status Bar	Toast	Speech	Dialog
1	Desactivado	Desactivado	Desactivado	Desactivado	Desactivado	Desactivado	Desactivado
2	Vibración Leve	Volumen Bajo	Parpadeo	Solo icono	Activado	Activado	Sólo Texto
3	Vibración Fuerte	Volumen Alto	Parpadeo con Patrón	Icono y Texto	-----	-----	Texto y Botones

El usuario podrá personalizar cada uno de los perfiles según sus necesidades, y los cambios que éste haga permanecerán guardados en el servidor al instante, entonces la aplicación volverá a leerlos la próxima vez que los necesite.

Servicios:

Representan las diferentes aplicaciones cuyas notificaciones se quieran personalizar, pudiendo para ello asociar un perfil a cada servicio. Por defecto vienen incluidas las Aplicaciones de serie del iPhone. Se podrán agregar más servicios en el servidor, tan sólo se tendrá que añadir el nombre del servicio en la base de datos y el icono de dicho servicio en el directorio ***/images/servicesIcons*** de nuestro servidor Apache. Dicho icono deberá ser una imagen en formato .png (con resolución aconsejada de 32x32 px) y con el mismo nombre del servicio incluido en la base de datos para que se muestre correctamente en la interfaz gráfica de la aplicación.

Transiciones entre perfiles:

Nuestra aplicación nos permite cambiar el perfil asociado a un servicio por otro perfil cuando se den una serie de condiciones. Al definir una transición entre perfiles podremos hacer que todos o parte de los servicios asociados a un determinado perfil cambien su perfil asociado por otro cuando se dé una o más de estas condiciones. Aportando gran flexibilidad a la personalización del grado de intrusión por parte del usuario.

Condiciones de las transiciones:

Las condiciones podrán ser creadas y editadas por el usuario, y estarán formadas por una o más sentencias con su respectivo sujeto, objeto y predicado. Estas sentencias serán del tipo:

> John WorksIn Office → Dividiendo la frase en tres partes tendremos, sujeto (subject), predicat y Object

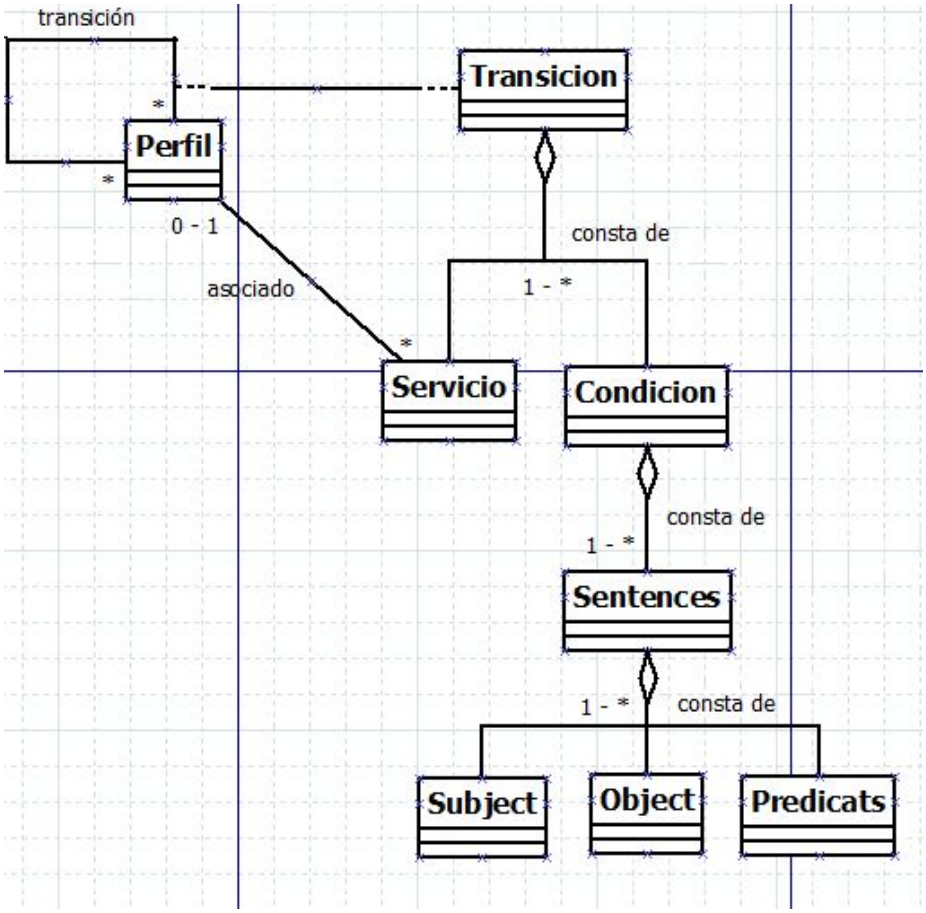
Las sentencias podrán ser creadas y configuradas por el usuario entre una serie de diferentes sujetos, objetos y predicados predefinidos en la aplicación, pudiendo añadir más al servidor si así se desea simplemente introduciendo nuevas entradas en la tabla de Subjects, Objects y Predicats, respectivamente, de nuestra base de datos. Además también podrá negar cada una de estas sentencias de manera individual para que represente justo lo contrario de lo que quiere decir la sentencia original.

Sentencias:

Las condiciones estarán formadas por uno o más elementos llamados sentencias. Los sujetos, objetos y predicados formarán cada sentencia. Estas sentencias están formadas por un tripla de Subject (sujeto), Object (Objeto) y Predicat (Predicado), haciendo referencia el subject y object a dos elementos (personas, objetos, lugares…) del sistema, mientras que el Predicat establece una relación entre estos elementos, es como una propiedad que une elementos. Estas sentencias describen un hecho que se da, y son el método con el que establecen las relaciones semánticas entre elementos de programación en el llamado OWL (Web Ontology Language), el lenguaje para publicar y compartir datos usando ontologías en la web.

Figura 14. Diagrama UML de objetos de la aplicación

A continuación dejamos un diagrama de modelado UML que se pretende seguir en la creación de esta aplicación, y que resume todo el comportamiento expuesto anteriormente:



3.1.2 Descripción de la arquitectura utilizada

Nuestra aplicación provee la infraestructura necesaria para la realización de todas estas tareas. Para su creación hemos elegido el modelo de programación **multicapa**. Este modelo ofrece tres niveles o capas en los que se divide la aplicación permitiendo así separar la parte de presentación de la lógica de datos. Estas capas son la capa de **presentación**, la capa de **datos** y la capa de **negocio.** Pasaremos a analizar en qué consiste cada una de ellas a continuación:

Por una parte tenemos la **capa de presentación,** que se trata de la parte de la aplicación correspondiente a la interfaz gráfica que se muestra al usuario. Mediante la cual éste interactúa, mostrándole la información necesaria y recogiendo los datos que se le solicitan cuando la aplicación así lo requiere, comprobando que no haya errores de formato. Esta capa interactúa únicamente con la capa de negocio, a través de la cuál recibe o transmite los datos que haya que mostrar al usuario o éste haya introducido a través de la interfaz.

La segunda capa corresponde a la **capa de negocio.** Es la encargada de analizar las acciones de usuario sobre la interfaz de la aplicación y actuar con una respuesta en consecuencia, es decir, recoge la lógica interna del funcionamiento de la aplicación. También es la encargada de interactuar con la capa de datos para recibir los datos que sean necesarios mostrar en la interfaz gráfica (capa de presentación) o almacenar en dicha capa de datos los datos que haya introducido el usuario durante el transcurso de la ejecución de aplicación. Haciendo así de **intermediaria** entre la capa de datos y la capa de presentación, proporcionándole a ésta última todos datos necesarios manteniendo el acceso a la parte de almacenamiento persistente de datos de manera transparente. De esta manera se cumplirá la separación por capas característica de esta arquitectura.

La tercera y última capa de esta arquitectura corresponde a la **capa de datos.** Su misión es la de almacenar de manera permanente los datos, garantizando su integridad (utilizando para ello una base de datos), y ofrecer los mecanismos necesarios para que la capa de negocio pueda acceder a ellos.

Con esta arquitectura conseguimos separar la aplicación en tres partes independientes, permitiendo hacer cambios en cada una de las capas sin que afecte a las demás capas. Pudiendo así interconectarlas solamente conociendo su interfaz de acceso, es decir, su respectiva API. Esta separación facilita el desarrollo por grupos.

Durante este capítulo analizaremos como hemos aplicado esta arquitectura en nuestra aplicación, señalando qué partes de ella componen cada capa y analizándolas en profundidad. En primer lugar trataremos la capa de presentación mostrando unas capturas de pantalla y explicaremos su usabilidad cara al usuario, después hablaremos de la capa de datos explicando como almacenamos nuestros datos de manera permanente -utilizando para ello un sistema de base de datos- y los mecanismos que se ofrecen para su acceso; y por último analizaremos la capa de negocio, donde hablaremos de cómo se transfieren los datos (desde la capa de datos) que necesite mostrar la capa de presentación o cómo se envían los datos introducidos por el usuario que se necesiten almacenar en la capa de datos de nuestra aplicación, así como una

explicación de la lógica más importante de las funciones de esta aplicación de personalización de perfiles de notificación según contexto de usuario.

Cabe destacar que para obtener los conceptos necesarios sobre la programación en iOS para realizar la implementación de la aplicación, me fue de gran ayuda el libro "*Objective-C: Curso práctico para programadores de MacOSX, iPhone y iPad*", de la editorial RC Libros, para familiarizarme y coger la soltura suficiente con el lenguaje Objective-C. También me ayudó bastante el libro "*Beggining iOS 5 Development*", de la editorial Apress, especialmente los temas de Views, Multi Views, TableViews, Navigation Controllers, Pickers, así como los temas iniciales para familiarizarse con la interfaz de Xcode y la creación de interfaces gráficas y conexión de objetos a dicha interfaz usando la herramienta Interface Builder. Con este libro aprendí todos los conceptos necesarios para la programación de aplicaciones iOS. También resultó útil la documentación que pone a disposición Apple con las especificaciones de las principales clases de iOS. Se adjuntan enlaces a toda esta documentación en el apartado de bibliografía y fuentes al final de esta memoria.

3.2 Arquitectura multicapa

Como hemos comentado, en este tema analizaremos las tres diferentes capas que componen nuestra aplicación: en primer lugar analizaremos la capa de presentación, donde ofreceremos capturas de pantalla de cada una de las vistas que componen nuestra aplicación tal y como se muestran éstas al usuario, explicando toda la funcionalidad que se puede extraer de ellas, dejando de lado la parte lógica de la aplicación. Es decir analizaremos el funcionamiento de aplicación tal y como la percibe un usuario de la misma.

En segundo lugar profundizaremos sobre la capa de datos, explicando cómo se almacenan los diferentes datos de nuestra aplicación en un sistema de base de datos SQL que hace posible su persistencia en el tiempo. Así como también hablaremos de los métodos de acceso a dichos datos, ya sea para lectura, escritura o modificación de las tablas, que nos ofrece esta capa mediante el uso de scripts PHP que se encargarán del acceso a las tablas de la base de datos.

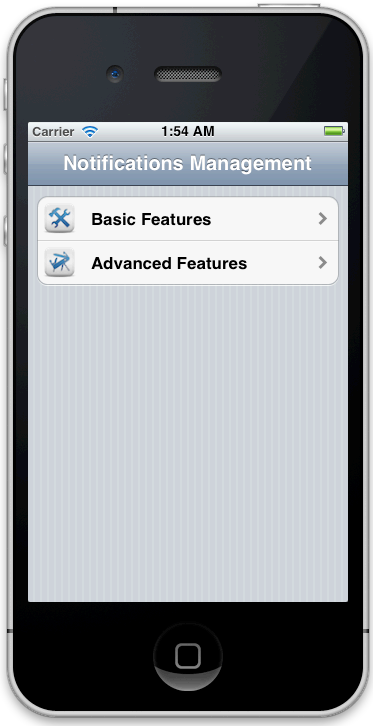
Por último lugar hablaremos de la capa que conecta y hace posible la comunicación entre las dos capas anteriores, la capa de negocio o capa lógica, analizando de manera detallada las principales partes lógicas de las funciones que ofrece nuestra aplicación y cómo se comunica esta capa con la capa de datos, para realizar los accesos de lectura, escritura o modificación que requiera la aplicación según las acciones que realice usuario. Para ello se utilizarán los citados scripts de acceso PHP que ofrece la capa de datos y la codificación JSON para la correcta interpretación de estos datos.

3.2.1 Capa de presentación

Esta capa representa la interfaz gráfica con la interactúa el usuario y mediante la cual podrá acceder a los datos del sistema, pudiendo leerlos, modificarlos o borrarlos de manera intuitiva y sencilla por medio de la interfaz. En este apartado explicaremos la información que muestra cada una de las pantallas de nuestra aplicación, su utilidad, así como los controles que componen cada pantalla y cual será su función. Nos centraremos en la utilidad que el usuario percibe al interactuar con nuestra aplicación, dejando de lado todas las cuestiones lógicas que hacen posible el flujo de información.

Cabe destacar que toda la información que se muestra en estas pantallas, se obtiene/almacena en la capa de datos por medio de la capa de negocio, actuando ésta última capa como intermediario y siendo la única capa con la que interactúa la capa de presentación para la lectura/almacenamiento de los datos en el sistema.

3.2.1.1 Menú principal



Esta es la primera pantalla que se encuentra el usuario al arrancar la aplicación. Podemos observar que ésta es una pantalla con una estructura simple cuyos principales elementos son una tabla (tableView) donde se muestran una serie de filas, conteniendo cada una de estas filas un icono a la izquierda que describe gráficamente la función de dicha fila, y otro icono a la derecha en forma de flecha que indica que al seleccionar esa fila nos llevará a otra vista en consecuencia.

En esta pantalla se muestran dos opciones entre las que el usuario podrá elegir: Características Básicas (Basic Features) y Características Avanzadas (Advanced Features). Si elige la primera opción podrá pasar a personalizar la asociación de los diferentes perfiles a cada una de las aplicaciones del sistema; mientras que si selecciona la segunda opción pasará a la sección de configuración avanzada donde podrá configurar los diferentes perfiles, las transiciones ente dichos perfiles y administrar las condiciones.

Figura 15. Configuración básica: **Asociación de un perfil a un servicio**

3.2.1.2 Configuración Básica

Si el usuario selecciona la primera opción, pasará a esta pantalla de configuración básica en la que se muestra una tabla que representará cada servicio del teléfono en cada una de sus filas. En cada fila se muestra el nombre e icono del servicio al que hace referencia, y debajo del nombre, en la parte de lo que se denomina subtítulo de la celda, encontramos el nombre del perfil asociado por defecto, en caso de tener uno.

Si el usuario selecciona una de estas filas, se abrirá una nueva ventana donde podrá elegir el perfil asociado al servicio seleccionado (2º imagen parte inferior) moviendo un deslizador (slider) que le permitirá cambiar entre los diferentes perfiles que haya definidos en el sistema. Se puede observar que por cada perfil existente se muestra una representación gráfica de los diferentes tipos y niveles de mecanismos de intromisión que tenga asociados el perfil seleccionado con el slider.

Podemos observar que a parte de la mencionada tabla, existe otro elemento importante en esta interfaz: un botón de la barra superior con el título "**More**". Si pulsamos este botón aparecerá una hoja de diálogo (**UIActionSheet**) por la parte inferior de la pantalla con tres opciones (3º imagen parte superior), "***Set all services***", "***Unset all services***" y "***Cancel***". Si el usuario selecciona la primera opción podrá asociar todos los servicios a un

mismo perfil, usando para ello la misma pantalla de selección de perfil vista anteriormente. Seleccionando la segunda opción podrá resetear la información, dejando todos los servicios sin perfil asociado. Mientras que si elige la tercera opción, Cancel, dicha hoja de diálogo desparecerá.

3.2.1.3 Configuración Avanzada

Si el usuario selecciona la segunda opción en el menú principal accederá a esta pantalla que consta de un submenú con varias opciones de configuración avanzada. Estas opciones son: **Profiles Configuration, Profiles Transitions y Manage Conditions**. Cuando el usuario selecciona una de las opciones pasará a la pantalla correspondiente a dicha configuración. Si se selecciona el primero de estos apartados el usuario podrá personalizar los perfiles, si elige el segundo pasará a la vista que le permite crear, eliminar y editar transiciones; mientras que si se selecciona la tercera opción, ésta nos llevará a una pantalla donde podremos crear, eliminar y editar condiciones. Analizaremos la funcionalidad de cada uno de estos apartados de configuración avanzada a continuación.

3.2.1.4 Configuración de un Perfil

En esta vista de nuestra aplicación el usuario puede personalizar cada uno de los perfiles existentes en el sistema según sus prioridades. Para ello, en primer lugar, se selecciona el perfil a configurar pulsando la fila de la sección *Profile to configure (*contendrá una fila con el nombre del perfil seleccionado actualmente para configurar*)* que mostrará una nueva ventana con un **Picker** que nos permitirá seleccionar el perfil. Una vez tenemos seleccionado un perfil a configurar, se cargará la información de dicho perfil en la vista, representando cada sección de la tabla a un mecanismo diferente de intrusión, y cada fila de estas secciones a un nivel de intrusión (molestia) diferente para dicho mecanismo. Se añade un 'tick' al lado del nivel de alerta asociado a cada mecanismo del perfil, y podremos editar el grado de intrusión de dicho perfil seleccionando otras filas. Al variar alguno de los mecanismos de intromisión, aparecen en la barra superior dos iconos a izquierda y derecha **Cancel** y **Done** que nos permitirán por un lado guardar los cambios y por otro cancelar todos los cambios realizados sobre ese perfil, restaurando su configuración inicial. Podemos observar capturas de este comportamiento en la parte inferior.

Figura 16. Configuración de un Perfil

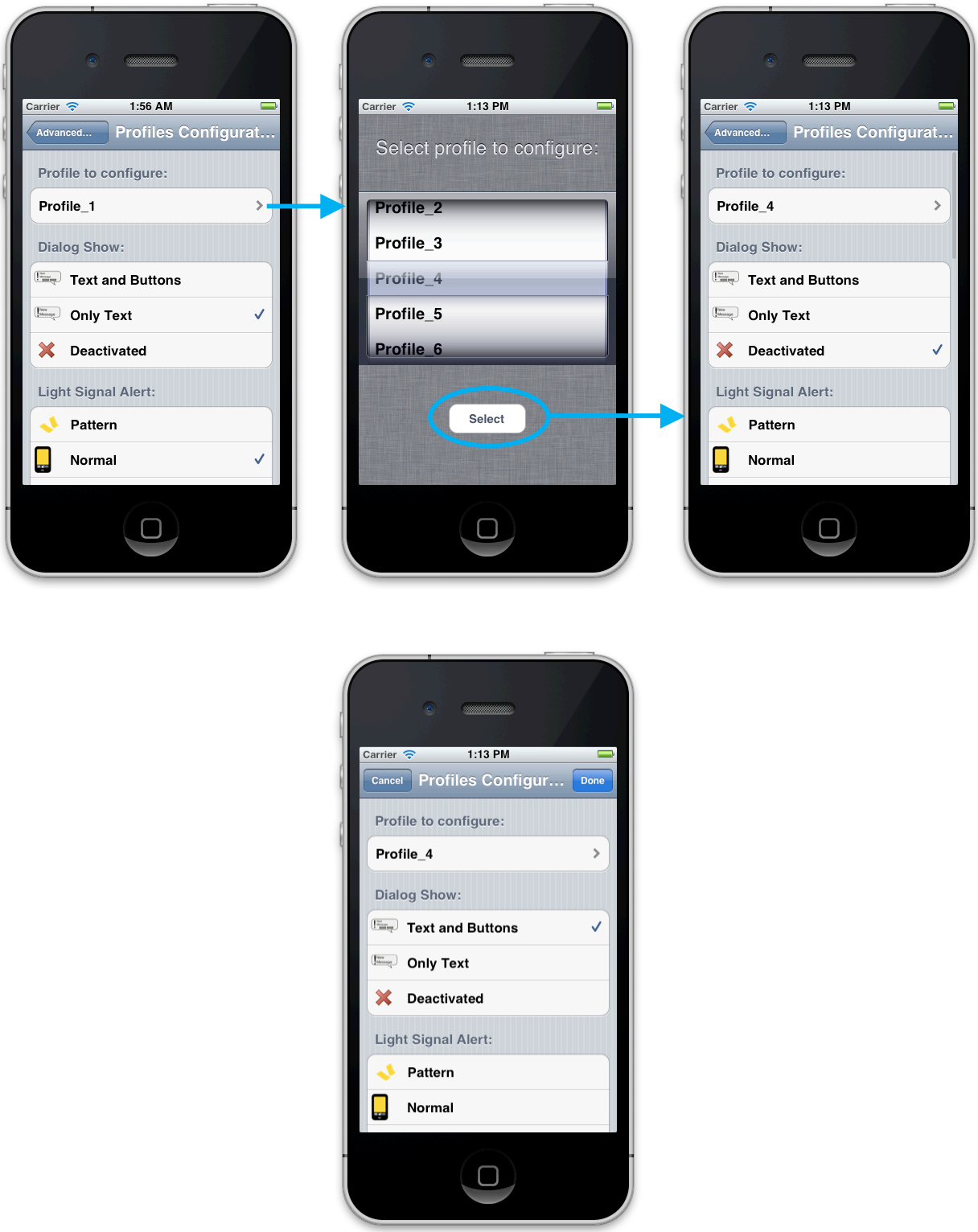


Figura 17. Selección entre las transiciones definidas en el sistema

3.2.1.5 Transiciones entre perfiles

Seleccionando la segunda opción de la configuración avanzada pasaremos la parte de nuestra aplicación donde se pueden definir, eliminar y modificar transiciones entre los perfiles asociados a los servicios o aplicaciones. La primera pantalla que aparece muestra una tabla con las transiciones que se encuentran definidas en ese momento. Se mostrará en el texto de cada fila el perfil origen y el perfil destino al que se cambiará, mientras que el subtítulo se indicará que servicios de los asociados al perfil original (**Perfil From**) se verá afectado en dicha transición. Observamos que existe un elemento botón en la parte derecha de la barra superior, si dicho botón se pulsa, el usuario podrá eliminar transiciones o crear una nueva.

Si se quiere modificar una transición, el usuario deberá seleccionar su fila correspondiente y se mostrará una pantalla que ofrece todos los mecanismos necesarios para su edición. Esta pantalla es compartida con el apartado de crear condición, con la única diferencia que en este último caso los datos de la interfaz estarán vacíos esperando la entrada del usuario en lugar de mostrar los datos correspondientes a la transición seleccionada. Veremos los detalles de esta vista a continuación:

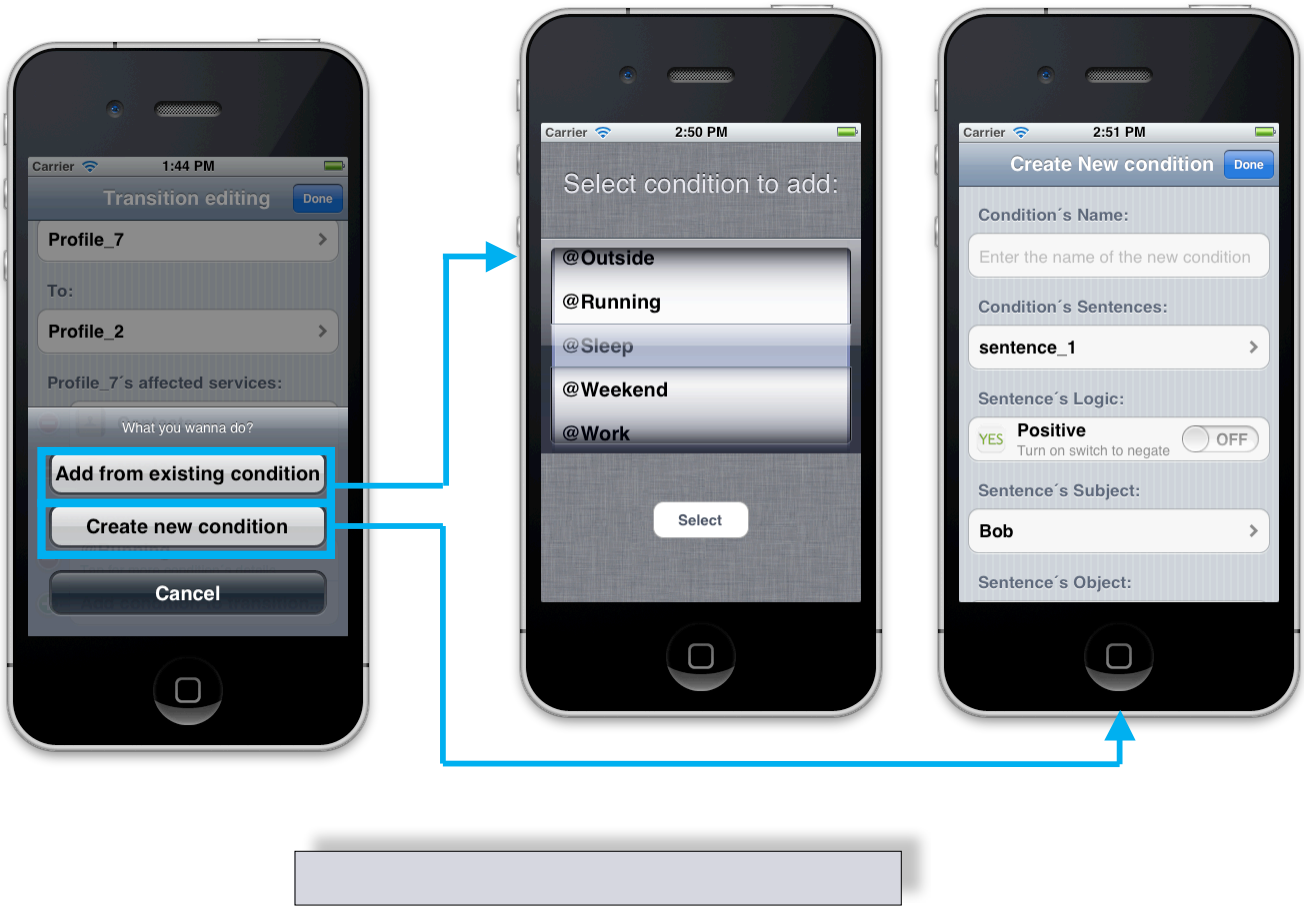
Figura 18. Interfaz de Creación/Edición de una transición

Esta es la interfaz que ofrece nuestra aplicación para la creación y modificación de transiciones, podemos observar que en un primer momento nos muestra (sin poder modificar) la información correspondiente a dicha transición, como son el Perfil **From** (perfil origen) y el Perfil **To** (perfil destino), entre los cuales se realizará la transición. Vemos también que en la tercera sección de filas se muestran qué **servicios** de los asociados al **Perfil From** se verán afectados en dicha transición. Mientras que la cuarta y última sección nos indica las **condiciones** que lanzarán la transición. Cuando el usuario pulse el botón **Edit,** se le permitirá modificar esta información (como se observa en la captura central de la parte superior) pudiendo seleccionar qué servicios del Perfil From se verán afectados (borrando filas de la sección servicios) y dando la posibilidad de borrar o añadir condiciones, pudiendo añadir bien una existente o una creada por el usuario en ese mismo instante.

Podremos también variar el perfil **From** y **To** pulsando en sus respectivas filas, esto es útil en la creación de transiciones que, como hemos comentado, se realiza también utilizando esta interfaz gráfica. Al variar el perfil From, se mostrarán los servicios asociados a dicho perfil seleccionado en la sección de servicios, y en caso de existir una transición entre los dos perfiles elegidos, se mostrarán los servicios afectados. Al pulsar el botón *Edit*, el usuario también podrá seleccionar las filas de la cuarta sección (*Conditions*), que lo llevarán a la pantalla de

creación/edición de condición mostrando los detalles de la condición seleccionada (pero no los podrá editar).

Pasaremos a explicar las pantallas correspondientes a las dos opciones que se le permiten seleccionar al usuario cuando añade una nueva condición a una transición, "Añadir una condición existente" o "Crear una nueva condición":



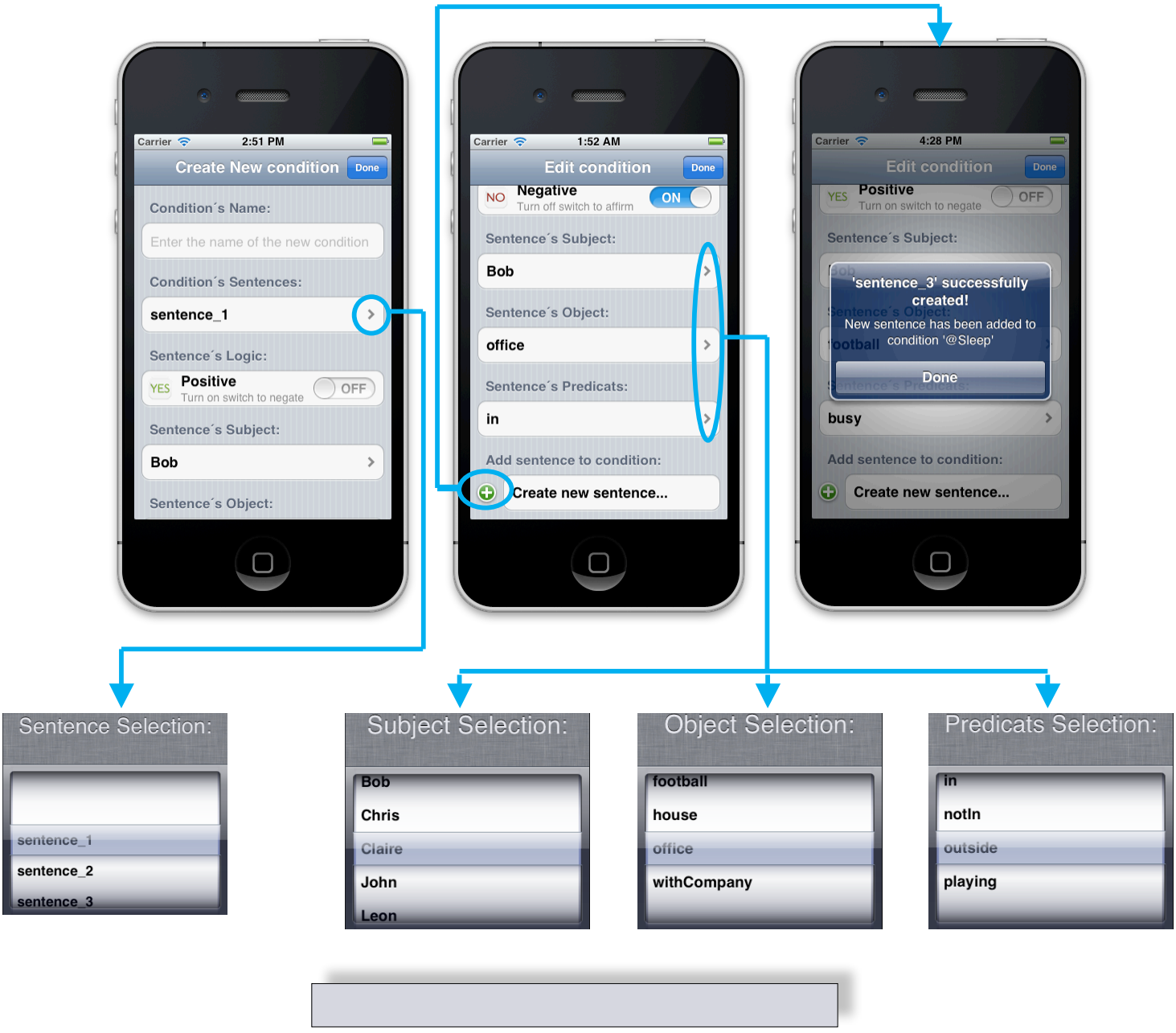
Cuando el usuario decide añadir una condición a una transición, ésta se añadirá al conjunto de condiciones de la transición si la transición existe, sino, dará paso a la creación de una nueva transición para los perfiles From y To elegidos. El sistema permite al usuario seleccionar la condición a añadir de dos maneras diferentes: añadiendo una de las existentes en el sistema, en cuyo caso se mostrará una vista con un **Picker** que hará posible la selección; o dando la posibilidad de **crear** una nueva condición que será añadida al sistema y a continuación se asociará a la respectiva transición.

Pasaremos a explicar toda la funcionalidad de la pantalla que nos permite crear o editar una condición.

Figura 20. Interfaz de Creación/Edición de una condición

3.2.1.6 Creación/Edición de condición

Esta es la interfaz que plantea nuestra aplicación para la creación y modificación de condiciones. Se muestra cuando el usuario desea crear una condición y cuando se requieren ver o editar los detalles de una determinada condición.



Nos permite por una parte introducir o modificar el nombre de la condición y por otra editar o añadir nuevas sentencias. Si se está creando una condición aparecerá un texto en el *TextField* de la interfaz asociado al nombre de la condición*: "Enter the name of the new condition"*. También podremos crear y editar las sentencias asociadas a la condición, para ello el usuario debe seleccionar la sentencia a modificar tocando la única fila de la sección de

nombre "**Condition´s Sentences",** se ofrecerá un Picker que permitirá seleccionar una sentencia de las que forman la condición a editar. Una vez elegida la sentencia que el usuario quiera modificar, se cargan sus valores originales en las filas de la UI las correspondientes a: lógica de la sentencia, subject, object y Predicat, pudiendo el usuario modificar estos valores a su antojo (utilizando para ello otros 3 pickers). También podrá crear una nueva sentencia para la condición pulsando para ello el botón (+) de la última fila. Al crear una nueva sentencia se mostrará un aviso, tal y como se observa en la tercera captura de la imagen superior.

3.2.1.7 Manage Conditions

Por último, cuando el usuario selecciona la tercera opción del menú de configuración avanzada **"Manage Conditions"** podrá administrar las diferentes condiciones usadas para crear las transiciones entre los diferentes perfiles de molestia. Siendo posible editar, borrar y crear nuevas condiciones.

Figura 21. Administración de condiciones definidas en el sistema

Esta pantalla se muestra como una tabla en cuyas filas se encuentra el nombre de cada una de las diferentes condiciones existentes. Si el usuario toca alguna de estas filas, se mostrarán los detalles sobre la condición seleccionada (imagen derecha) y serán editables por el usuario, pudiendo cambiar el nombre y modificar/crear las sentencias que la componen. También existe un botón *Edit* en la barra superior de la pantalla que permitirá borrar o crear condiciones al pulsarlo (imagen central). Si el usuario decide crear una nueva condición, aparecerá la pantalla vista anteriormente de creación de condición, que ofrecerá la infraestructura necesaria para ello.

3.2.2 Capa de datos

Esta capa es la encargada de almacenar todos los datos que utiliza la aplicación de manera persistente en el tiempo. Se encarga de guardar los cambios que haga el usuario (condiciones borradas, transiciones creadas…), que se mantendrán entre las diferentes sesiones de uso de la aplicación. Estos datos se almacenan utilizando una base de datos MySQL y son accedidos a través de la capa de negocio, haciendo así posible la interacción del usuario con estos datos por medio de la interfaz gráfica de la aplicación. Para el intercambio de datos entre la capa de negocio y la capa de datos, ésta última ofrece unos mecanismos de acceso por medio de una serie de scripts PHP. Estos scripts son utilizados para enviar información desde las tablas de la base de datos hacia la aplicación para que se muestren correctamente en las pantallas, o para recibir los datos introducidos por el usuario y enviarlos a dicha BD para su correcto almacenamiento. Dichos datos viajan utilizando un sistema de codificación llamado JSON, que hace posible que la aplicación entienda y pueda interpretar los datos recibidos desde la base de datos.

En el anexo de esta memoria se puede encontrar una descripción detallada de la instalación y puesta a punto del software y herramientas utilizados en la parte servidor de la aplicación y que hace posible toda esta infraestructura de comunicación cliente-servidor.

A lo largo de este capítulo explicaremos en primer lugar cuál la estructura de nuestra base de datos: indicando qué atributos tiene cada tabla, qué representa cada uno dentro del contexto de la aplicación y las restricciones de integridad definidas entre tablas (claves ajenas, claves primarias…). Una vez se conozca el esquema básico, pasaremos a comentar los scripts escritos en PHP e incluidos en el servidor web creado que hacen posible el acceso a estas tablas de la base de datos, ya sea para leer, borrar, crear o modificar datos, y que son utilizados desde la capa de negocio de la aplicación por medio de peticiones HTTP.

3.2.2.1 Base de datos MySQL

MySQL es el nombre del sistema de gestión de base de datos utilizado en este proyecto. He elegido esta opción debido a que es un sistema muy extendido y con cual estoy familiarizado. Gracias a esta herramienta podremos hacer que todos los cambios que realice el usuario sobre los datos de la aplicación permanezcan almacenados de manera persistente en el tiempo, pudiendo se recuperados entre diferentes ejecuciones de la aplicación o incluso por diferentes dispositivos que ejecuten la aplicación.

Para almacenar todos los datos necesarios para el correcto funcionamiento de la aplicación se han creado nueve tablas de datos, que son las siguientes: Servicios, Perfiles, Condiciones, Exchanges, Subject, Object, Predicat, ServiciosTransiciones y SentenciasCondiciones. En este punto analizaremos todos los atributos que almacenan dichas tablas: qué representan dentro del entorno de la aplicación, que relaciones entre ellos existen y qué restricciones de integridad cumplen para asegurar la correcta persistencia e integridad de los datos en el transcurso del tiempo:

Tabla Perfiles:

Cada fila de esta tabla representa un perfil de molestia diferente. Tiene 8 atributos: el primero de ellos corresponde al **nombre del perfil** y actúa como clave primaria de la tabla, mientras que los otros 7 restantes hacen referencia a los **diferentes mecanismos de notificación** del dispositivo y pueden almacenar el valor 1, 2 o 3 que indicará el nivel de alerta del mecanismo correspondiente.

Tabla Servicios:

Esta tabla almacena los servicios de la aplicación, consta de 2 atributos: el primero de ellos actúa de clave primaria y es el **nombre del servicio**, mientras que el segundo corresponde al nombre del **perfil asociado** a dicho servicio y define una restricción de clave ajena con el atributo nombre perfil de la tabla *perfiles*.

Tabla Condiciones:

Esta tabla contiene el nombre de todas las condiciones que existen en el sistema en ese momento. Está formada únicamente por un atributo: **nombre**, que a su vez actúa como clave primaria de la tabla.

Tablas Subjects, Objects y Predicats:

Estas tres tablas almacenan los Subject, Objects y Predicats, respectivamente, que serán utilizados en la creación de las sentencias que a su vez formarán las condiciones. Estas tablas constan de un solo atributo, **nombre**, que hace referencia al identificador de cada Subject, Object y Predicat. Este atributo a su vez jugará el papel de clave primaria en cada tabla.

Tabla SentenciasCondiciones:

Cada una de filas de esta tabla representa una sentencia asociada a una de las condiciones definidas en el sistema. Consta de cinco atributos: **nombre**, que hace referencia al nombre de la condición a la que pertenece dicha sentencia y cumple restricción de clave ajena con la tabla *condiciones*; **subject, object y Predicat**, que hacen referencia a los respectivos sujetos, objetos y predicados de la sentencia y cumplen un restricción de clave ajena con el atributo nombre las respectivas tablas (*Subjects*, *Objects* y *Predicats*); mientras que el último atributo **negada**, indica si la sentencia tiene lógica afirmativa o negativa. La clave primaria de esta tabla, que identificará de manera única a cada una de las sentencias, estará definida por los cuatro atributos: nombre, subject, object y Predicat.

Tabla Exchanges:

Esta tabla almacena las condiciones que lanzarán las **transiciones entre perfiles** que ha creado el usuario. Consta de tres atributos: **Perfil From y Perfil To,** que indican el perfil origen y destino de la transición, cumpliendo ambos una restricción de clave ajena con el atributo nombre de la tabla *Perfiles*; y el atributo **nomb_condicion** que indica el nombre de la condición asociada a dicha transición y cumplirá a su vez una restricción de clave ajena con el atributo nombre de la tabla *Condiciones*. Existirá una tupla en esta tabla por cada una de las

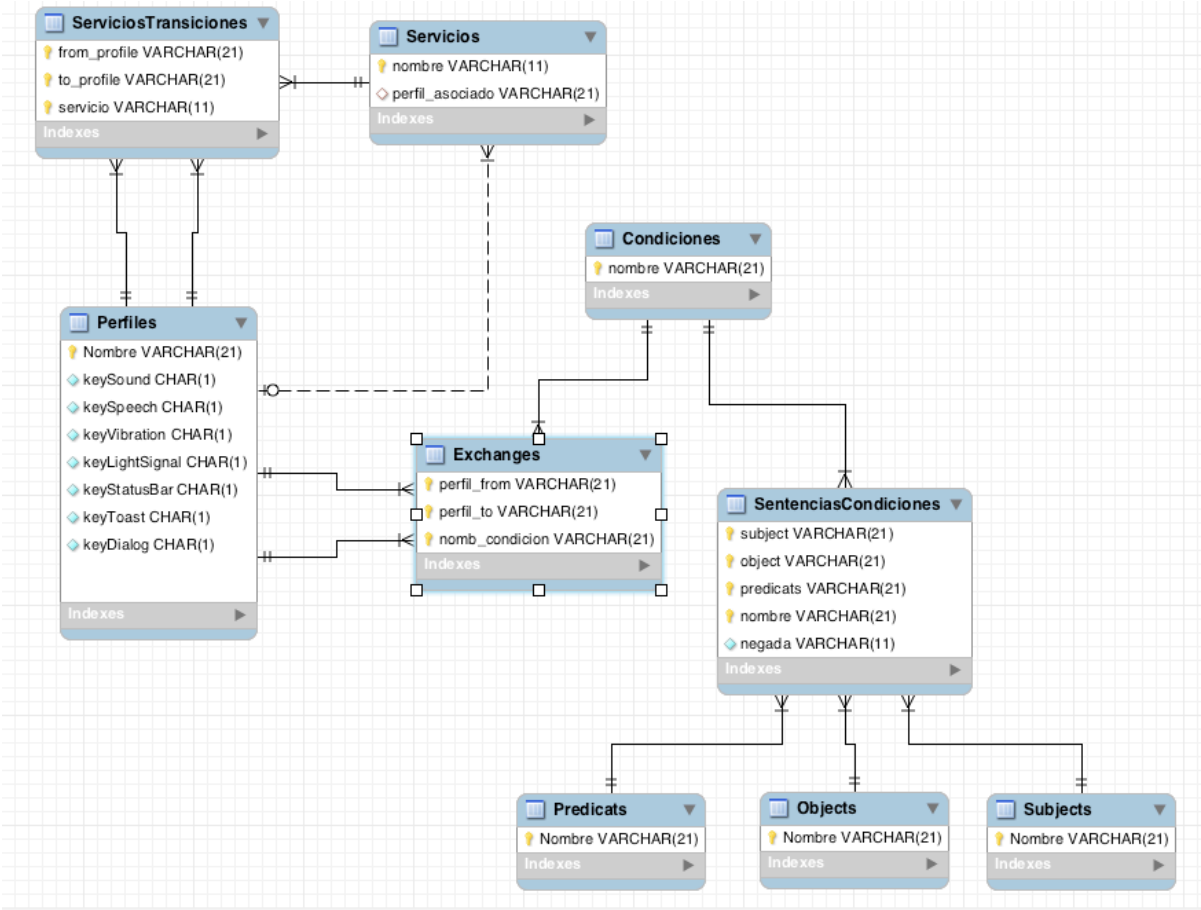
Figura 22. Esquema de tablas de la base de datos MySQL

condiciones que formen una transición, por lo que sus tres atributos actúan como clave primaria de la tabla.

Tabla ServiciosTransiciones:

Esta última tabla contiene una fila por cada servicio que se vea afectado por una transición. Dispone de tres atributos: **Perfil From y Perfil To,** que indican el perfil origen y destino de la transición, cumpliendo ambos una restricción de clave ajena con el atributo nombre de la tabla Perfiles; y el atributo **servicio** que señalará al servicio afectado en la transición, viéndose afectado este atributo también por una restricción de clave ajena hacia el atributo nombre de la tabla Servicios. La clave primaria de esta tabla corresponde a la tripla formada por sus tres atributos.

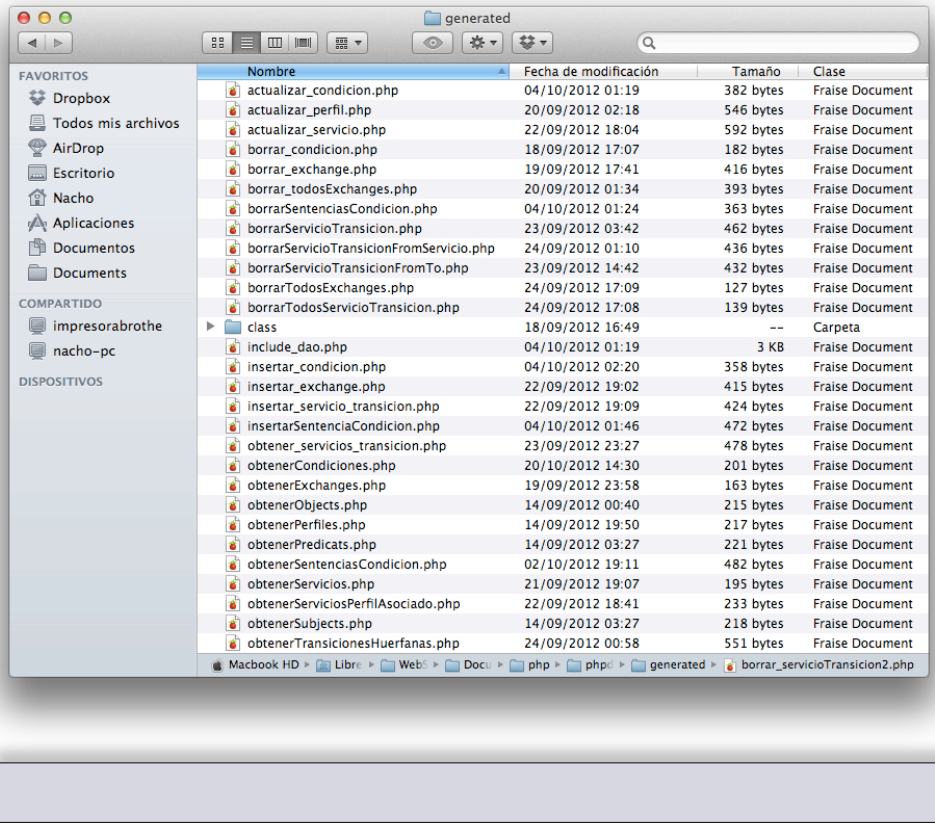
Este es el esquema de tablas de nuestro sistema de persistencia de datos, gracias a él y a las reglas que define, podemos garantizar la integridad y consistencia de los datos. A continuación se muestra un resumen de esta base de datos creado con la herramienta *MySQL WorkBench*. Se muestran los diferentes atributos y su tipo, así como las restricciones de clave ajena (representadas por flechas) creadas todas ellas con **borrado/modificación en cascada.** También se indican las **claves primarias** de cada tabla mediante un icono en forma de llave a la izquierda del nombre del respectivo atributo/s:



3.2.2.2 Scripts PHP para acceso a la capa de datos

El método de acceso a estos datos desde la aplicación que pone a nuestra disposición esta capa de datos se realiza utilizando scripts PHP, que accederán a las diferentes tablas del SGBD. Para ello se harán peticiones HTTP (REST) desde la capa de negocio de la aplicación -con sus respectivos argumentos si es necesario- al script que se tenga que ejecutar en cada momento dependiendo de la tarea que deba realizar sobre la base de datos según las acciones realizadas por el usuario.

La aplicación requiere cuatro tipos de accesos diferentes a las tablas: lectura, escritura, borrado y modificación. Para llevar a cabo cada una de estas operaciones hemos utilizado por una parte los métodos de acceso de las clases generadas por la herramienta PHP DAO Generator (de la que se habla ampliamente en el anexo), mientras que para realizar consultas más concretas se ha optado por la creación manual de scripts que hagan sus respectivas consultas sql (querys) a la base de datos.



En el transcurso de este apartado veremos un ejemplo de un script para cada uno de los cuatro tipos de acceso a la BD, del que analizaremos su funcionamiento y explicaremos los cambios o consultas que realiza en las tablas. Una vez hecho esto tendremos una idea clara de como funcionan todos los scripts de acceso, ya que de unos a otros solo cambia las tablas que consultan y obviamente los datos que solicitan, crear o modifican, pero su estructura es la misma. Podemos observar el contenido del directorio **/documents/php/phpdao/generated** en la imagen superior, en esta carpeta del servidor se encuentran todos los scripts que utiliza la

Figura 24. Scripts PHP de lectura de datos (Objects)

aplicación durante su ejecución, serán llamados por ésta (por medio de la llamada capa de negocio) haciendo peticiones HTTP e incluyendo, si así se requiere, los argumentos de la consulta SQL dentro de la URL de la petición. Se puede apreciar fácilmente, mirando los nombres de los ficheros, que existen cuatro tipos de tipos accesos diferentes a los datos de las tablas, estos son: actualizar, borrar, insertar y obtener. Pasaremos a exponer un ejemplo que de cada uno de estos tipos de acceso a continuación:

Lectura de datos:

Este es el primer caso de acceso a la BD que realiza nuestra aplicación. Se requieren leer los datos de las tablas para mostrar la información en las diferentes pantallas de la aplicación. Este script recibirá argumentos si se desea aplicar un filtro en las filas de la respuesta devolviendo solo algunos de sus atributos. Para este tipo de accesos se utiliza el método **QueryAll()** implementado en la clase que genera DAO Generator para cada una de las tablas del sistema. DAO Generator crea una clase y una interfaz por cada tabla del SGBD que nos proporciona acceso a todos los métodos necesarios para realizar consultas básicas a las tablas. A continuación se muestra el código de un script de un acceso de este tipo, y el código del método QueryAll() que utiliza y hace posible la lectura de datos:

```php
<?php
// Include all DAO files
require_once('include_dao.php');

$transaction = new Transaction();
$todosObjects = DAOFactory::getObjectsDAO()->queryAll();
echo json_encode($todosObjects);
$transaction->commit();
?>
```

```php
    public function queryAll(){
        $sql = 'SELECT * FROM Objects';
        $sqlQuery = new SqlQuery($sql);
        return $this->getList($sqlQuery);
    }
```

Se puede observar que en primer lugar se incluye el fichero de interfaz (*include_dao.php*) generado para poder utilizar las librearías y métodos de acceso DAO, esto será necesario en todos los scripts que utilicen clases DAO. Después se llama al método **QueryAll()**, implementado en la clase *ObjectsMySqlDAO.class.php* (generada por DAO), correspondiente a la tabla *Objects,* este método listará todas las filas de dicha tabla. El resultado devuelto por dicho método se guarda en la variable **$todosObjects** que será codificada en formato JSON usando el método **json_encode()** y escrita en la página web devuelta a la aplicación mediante la instrucción **echo.** Esto hará posible la comunicación entre ambas capas como si la visualización de una página web se tratara.

Escritura de datos:

Este es el acceso que se da cuando en la aplicación el usuario crea una condición, transición, asocia un perfil a un servicio, etc. En estos casos será necesario insertar una tupla en alguna de las tablas de la base de datos. Este tipo de acceso requerirá incluir los argumentos a insertar en la BD dentro de la URL con la que se invocan los scripts, para después añadirlos a la consulta SQL. Se muestra un ejemplo a continuación:

```php
<?php
$con = mysql_connect("127.0.0.1","root","123456");
if (!$con)
{
  die('No se puede conectar a la BD Mysql: ' . mysql_error());
}

mysql_select_db("PFC", $con);
$sql="INSERT INTO Exchanges (perfil_from, perfil_to, nomb_condicion)
VALUES('$_GET[from]','$_GET[to]','$_GET[nomCond]')";

if (!mysql_query($sql,$con))
{
  die('Error: ' . mysql_error());
}

echo "Fila anyadida correctamente";

mysql_close($con);
?>
```

Para la creación de este tipo de accesos hemos prescindido de la herramienta DAO, en su lugar hemos realizado las consultas utilizando la librería sql incluida en PHP. Estos scripts reciben argumentos dentro de la variable asociada al método GET, que corresponderán a los datos que formen la tupla a añadir. En este caso se añade una condición a una transición, que requerirá como argumentos el perfil From, perfil To y el nombre de la condición. Estos datos habrán sido enviados por la capa de negocio a través de una petición HTTP (incluidos dentro de la URL) hacia un objeto .php. Se puede observar en el código como se intenta conectar a la base de datos y se avisa al usuario en caso de error en la conexión o durante la realización de la consulta. La consulta "INSERT", formada a partir de los argumentos recibidos en la variable GET, se guarda en una variable **$sql** y es ejecutada utilizando el método **mysql_query($sql, $con),** que recibe como argumentos la consulta (query) y la conexión a la base de datos realizada previamente. Una vez ejecutada la consulta, se habrá añadido la fila en la tabla correspondiente.

Figura 26. Script PHP de modificación de datos (Condición)

Modificación de datos:

Para la modificación de datos se ha seguido el mismo esquema que para la escritura. Este tipo de acceso requiere dos o más argumentos que contendrán los nuevos datos a modificar y servirán para identificar la fila afectada. Dicho acceso será necesario cuando el usuario modifique el valor de algún atributo de las tablas, ya sea porque actualice un perfil, modifique una condición, asocie un perfil a un servicio…

A continuación dejamos una captura de pantalla que representa la modificación del nombre de una condición, lo que implicará que se tendrá que modificar una fila de la tabla *condiciones* del SGBD.

```php
<?php
$con = mysql_connect("127.0.0.1","root","123456");
if (!$con)
{
  die('No se puede conectar a la BD Mysql: ' . mysql_error());
}

mysql_select_db("PFC", $con);

$sql="UPDATE Condiciones SET nombre='$_GET[nombre]' WHERE nombre='$_GET[nombreOrig]';";

if (!mysql_query($sql,$con))
{
  die('Error: ' . mysql_error());
}

echo "Fila añadida correctamente";

mysql_close($con);
?>
```

El código es similar al del caso de escritura salvo porque se cambia el tipo consulta por "UPDATE". En este ejemplo lo que se hace es actualizar una fila de la tabla condiciones, en concreto modifica el *nombre* de la tupla cuyo valor del atributo *nombre* sea igual al valor de la variable GET[nombreOrig], recibida como argumento del script. Esta modificación se repercutirá en cascada a las respectivas tablas que tengan restricción de clave ajena con este atributo *nombre* de la tabla condiciones, como *SentenciasCondiciones* o *Exchanges*. Garantizando así la consistencia de los datos.

Borrado de datos:

Durante el transcurso de ejecución de la aplicación es posible que el usuario decida borrar alguna condición, transición o algún otro elemento del sistema, por lo que será necesario ofrecer un mecanismo para la eliminación de tuplas de las tablas desde la aplicación. Para la realización de estos scripts de borrado se ha utilizado tanto métodos de las clases creadas por DAO, como scripts creados 'a mano'. En la parte inferior se muestra un ejemplo de cada caso: El primero borra todas las filas de tabla Exchanges llamando para ello al método **clean()** de la clase asociada a la tabla Exchanges generada por DAO (implementada en el fichero *ExchangesMySqlDAO.class.php*). Se adjunta también el código de dicho método:

```php
<?php
// Include all DAO files
require_once('include_dao.php');

$condicionBorrada = DAOFactory::getExchangesDAO()->clean();
?>
```

```php
public function clean(){
    $sql = 'DELETE FROM Exchanges';
    $sqlQuery = new SqlQuery($sql);
    return $this->executeUpdate($sqlQuery);
}
```

Mientras que en el segundo ejemplo se borran ciertas tuplas de la tabla *SentenciasCondiciones* mediante una consulta (query) de tipo “DELETE”. Se borrarán aquellas sentencias asociadas a la condición cuyo nombre coincida con el recibido por el script como argumento mediante el método GET.

```php
<?php
$con = mysql_connect("127.0.0.1","root","123456");
if (!$con)
{
  die('No se puede conectar a la BD Mysql: ' . mysql_error());
}

mysql_select_db("PFC", $con);
$sql="DELETE FROM SentenciasCondiciones WHERE nombre='$_GET[nombre]'";

if (!mysql_query($sql,$con))
{
  die('Error: ' . mysql_error());
}

echo "Fila Borrada correctamente";

mysql_close($con);
?>
```

3.2.3 Capa de negocio

La capa de negocio es la que contiene toda la lógica de las tareas que realiza nuestra aplicación. A lo largo de este apartado hablaremos de la parte lógica más importante de estas funciones, así como los métodos con los que esta capa recupera información de la base de datos para que sea mostrada en la aplicación a través de la interfaz, o envía datos introducidos por el usuario que se deban guardar en la BD. Explicaremos las diferentes tareas que dan lugar al intercambio de datos entre la capa de negocio y la capa de datos, a través de scripts PHP, así como la lógica principal de las tareas más importantes que realiza la aplicación.

3.2.3.1 Lectura de datos

Es necesario que la aplicación se conecte a la base de datos nada más iniciarse para recuperar la información que en sus vistas se mostrará. La mayoría de datos que se necesitan son leídos al inicio, en la pantalla de *Menú Principal,* donde a partir de ellos se inicializan los objetos internos de la aplicación. Una vez leídos los datos no vuelve a ser necesario acceder de nuevo a las tablas hasta que no se modifique/inserte/borre algún elemento del sistema. Explicaremos dónde se producen estos accesos, qué métodos los llevan a cabo y qué información recuperan:

Lectura de Perfiles, Subjects, Objects, Predicats y Condiciones:

Estos datos se leen nada más empezar, utilizando para ello una serie de métodos que se ejecutarán al iniciar la aplicación, concretamente dentro del método *-(void)viewDidLoad* de la vista del menú principal. Con ellos se inicializan parte de los datos que la aplicación necesitará mostrar en las siguientes pantallas. Estos métodos de lectura recibirán como argumentos la información devuelta por la peticiones HTTP realizadas a las URLs de los scripts PHP correspondientes, que como hemos explicado con anterioridad vendrá codificada mediante JSON y será enviada a través de un objeto tipo NSDATA (*datos)* a dichos métodos.

```
#define urlPerfiles [NSURL URLWithString: @"http://localhost/php/phpdao/generated/obtenerPerfiles.php"]
#define urlGenerate [NSURL URLWithString: @"http://localhost/php/phpdao/generate.php"]
#define urlObjects [NSURL URLWithString: @"http://localhost/php/phpdao/generated/obtenerObjects.php"]
#define urlPredicats [NSURL URLWithString: @"http://localhost/php/phpdao/generated/obtenerPredicats.php"]
#define urlCondiciones [NSURL URLWithString: @"http://localhost/php/phpdao/generated/obtenerCondiciones.php"]
#define urlSubjects [NSURL URLWithString: @"http://localhost/php/phpdao/generated/obtenerSubjects.php"]



- (void) viewDidLoad {
NSData *dataPerfiles = [NSData dataWithContentsOfURL: urlPerfiles];
NSData *dataSubjects = [NSData dataWithContentsOfURL: urlSubjects];
NSData *dataObjects = [NSData dataWithContentsOfURL: urlObjects];
NSData *dataPredicats = [NSData dataWithContentsOfURL: urlPredicats];
NSData *dataCondiciones = [NSData dataWithContentsOfURL: urlCondiciones];

      …

[self performSelectorOnMainThread:@selector(leerPerfiles:) withObject:dataPerfiles waitUntilDone:YES];
[self performSelectorOnMainThread:@selector(leerSubjects:) withObject:dataSubjects waitUntilDone:YES];
[self performSelectorOnMainThread:@selector(leerObjects:) withObject:dataObjects waitUntilDone:YES];
[self performSelectorOnMainThread:@selector(leerPredicats:) withObject:dataPredicats waitUntilDone:YES];
[self performSelectorOnMainThread:@selector(leerCondiciones:) withObject:dataCondiciones waitUntilDone:YES];
```

Observamos en el código que primero se definen una serie de constantes que almacenan las URLs de los scripts que leerán los datos de las diferentes tablas. En segundo lugar obtenemos los datos que devuelven las peticiones HTTP a dichas URLs utilizando para ello el método inicializador de la clase NSData: *dataWithContentsOfURL*. En este punto tendremos cinco variables que contendrán los datos devueltos por las respectivas tablas codificados mediante JSON y que serán usados (variable *datos*) para llamar a los métodos: leerPerfiles, leerSubjects, leerObjects, leerPredicats y leerCondiciones, respectivamente, que interpretarán los datos obtenidos tal y como se muestra en la siguiente parte de código correspondiente a cada uno de los métodos de lectura:

- **Perfiles**

```
- (void)leerPerfiles:(NSData *)datos {

    //////      Obtener Perfiles del servidor JSON      //////
    NSError* error;
    NSArray* perfilesLeidos = [NSJSONSerialization JSONObjectWithData:datos options:kNilOptions error:&error];

     ...

  for (int j = 0; j < [perfilesLeidos count]; j++) {
      NSMutableDictionary *perfilActual = [[NSMutableDictionary alloc] init];
      perfilActual = [[NSMutableDictionary alloc] initWithDictionary:[perfilesLeidos objectAtIndex:j]];
      [self.todosPerfiles setObject:perfilActual forKey:[self.nombresPerfiles objectAtIndex:j]];
  }
```

Se puede observar que el método recibe un argumento *datos de tipo *NSData*, este objeto contendrá la información leída de la tabla *Perfiles* codificada en JSON a partir del script. Esta información será interpretada por la capa lógica a través de la librería **NSJSONSerialization**, que nos ofrece el SDK de iOS (a partir de la versión 5) para tal fin. Una vez interpretados los datos, serán guardados en un objeto de colección NSArray que será recorrida, asociando cada uno de sus elementos (que constarán de un **NSDictionary**) a un perfil que será almacenado finalmente dentro de la colección los objetos que forman el sistema lógico.

- **Subjects**

```
- (void)leerSubjects:(NSData *)datos {
    NSMutableArray *todosSubjectsAux = [[NSMutableArray alloc] init];

    NSError* error;
    NSArray* datosLeidos = [NSJSONSerialization JSONObjectWithData:datos options:kNilOptions error:&error];
    for (int i = 0; i < [datosLeidos count]; i++) {
        NSMutableDictionary *datosActuales = [[NSMutableDictionary alloc] initWithDictionary:[datosLeidos objectAtIndex:i]];
        [todosSubjectsAux addObject:[datosActuales objectForKey:@"nombre"]];
    }
    self.todosSubjects = [[NSArray alloc] initWithArray:todosSubjectsAux copyItems:YES];

      ...
```

- **Objects**

```
- (void)leerObjects:(NSData *)datos
{
    NSMutableArray *todosObjectsAux = [[NSMutableArray alloc] init];

    NSError* error;
    NSArray* datosLeidos = [NSJSONSerialization JSONObjectWithData:datos options:kNilOptions error:&error];
    for (int i = 0; i < [datosLeidos count]; i++) {
        NSMutableDictionary *datosActuales = [[NSMutableDictionary alloc] initWithDictionary:[datosLeidos objectAtIndex:i]];
        [todosObjectsAux addObject:[datosActuales objectForKey:@"nombre"]];
    }
    self.todosObjects = [[NSArray alloc] initWithArray:todosObjectsAux copyItems:YES];
```

- **Predicats**

```
- (void)leerPredicats:(NSData *)datos
{
    NSMutableArray *todosPredicatsAux = [[NSMutableArray alloc] init];

    NSError* error;
    NSArray* datosLeidos = [NSJSONSerialization JSONObjectWithData:datos options:kNilOptions error:&error];
    for (int i = 0; i < [datosLeidos count]; i++) {
        NSMutableDictionary *datosActuales = [[NSMutableDictionary alloc] initWithDictionary:[datosLeidos objectAtIndex:i]];
        [todosPredicatsAux addObject:[datosActuales objectForKey:@"nombre"]];
    }
    self.todosPredicats = [[NSArray alloc] initWithArray:todosPredicatsAux copyItems:YES];
```

Estos tres objetos son almacenados en el sistema de la misma forma, dentro de un objeto de colección **NSArray.** Esto implica que el proceso de lectura siga el mismo esquema para los tres casos: En primer lugar se decodifica el argumento *datos* que recibe dicho método de las respectivas tablas (Subjects, Objects y Predicats) usando el la librería NSJSONSerialization y finalmente se almacena el nombre del objeto leído en cada uno de los respectivos objetos NSArray asociados, *todosSubjects*, *todosObjects* y *todosPredicats. S*erán tres Arrays definidas de manera inmutable ya que estos elementos no variarán durante el transcurso de la ejecución de la aplicación.

- **Condiciones**

```
- (void)leerCondiciones:(NSData *)datos
{
    NSError *error;
    NSArray* datosLeidos = [NSJSONSerialization JSONObjectWithData:datos options:kNilOptions error:&error];

    for (int k = 0; k < [datosLeidos count]; k++) {
        NSDictionary *condicionLeida = [[NSDictionary alloc] init];
        PFCObjetoCondicion *condicionActual = [[PFCObjetoCondicion alloc] init];
        condicionLeida = [datosLeidos objectAtIndex:k];
        condicionActual.nombreCondicion = [condicionLeida objectForKey:@"nombre"];

        NSString *getSentenciasCondicion = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated/
            obtenerSentenciasCondicion.php?nombre=%@",[condicionLeida objectForKey:@"nombre"]];
        NSURL *urlSentenciasCondicion = [NSURL URLWithString: getSentenciasCondicion];
        NSData *dataSentenciasCondicion = [NSData dataWithContentsOfURL: urlSentenciasCondicion];
        NSArray *sentenciasCondicion = [NSJSONSerialization JSONObjectWithData:dataSentenciasCondicion options:kNilOptions error:
            &error];

          ...

       NSMutableDictionary *sentencesAuxiliar = [[NSMutableDictionary alloc] init];
       //Añadimos a la condicion sus sentencias, un object, subject y predicat por cada sentencia
       for (int j = 0; j < [sentenciasCondicion count]-1; j++) {
           NSMutableDictionary *sentenciaLeida = [[NSMutableDictionary alloc] init];
           NSMutableDictionary *sentenciaAnyadida = [[NSMutableDictionary alloc] init];
           sentenciaLeida = [sentenciasCondicion objectAtIndex:j];
           [sentenciaAnyadida setObject:[sentenciaLeida objectForKey:@"object"] forKey:@"object"];
           [sentenciaAnyadida setObject:[sentenciaLeida objectForKey:@"subject"] forKey:@"subject"];
           [sentenciaAnyadida setObject:[sentenciaLeida objectForKey:@"predicats"] forKey:@"predicats"];
           [sentenciaAnyadida setObject:[sentenciaLeida objectForKey:@"negada"] forKey:@"negada"];
           NSString *keySentencia = [[NSString alloc] initWithFormat:@"sentence_%@", [[NSNumber alloc] initWithInt:j+1]];

           [sentencesAuxiliar setObject:sentenciaAnyadida forKey:keySentencia];
       }
       condicionActual.sentences = [[NSMutableDictionary alloc] initWithDictionary:sentencesAuxiliar copyItems:YES];

       [self.todasCondiciones setObject:condicionActual forKey:condicionActual.nombreCondicion];
    }
}
```

Para la el almacenamiento de las condiciones dentro de la aplicación se ha creado una clase llamada **PFCObjetoCondicion,** que constará de dos atributos: un **NSString** que representará el nombre de la condición y un **NSMutableDictionary** que almacenará todas sus sentencias.

Si nos fijamos en el código del método de la parte superior, observamos que recibe un argumento -*datos*- que contendrá el nombre (en JSON) de todas las condiciones existentes en el sistema (leídas de la tabla *Condiciones*). El código de este método se puede dividir en dos partes bien diferenciadas: en la primera de ellas se crea un objeto de tipo PFCObjetoCondicion que representará cada una de las condiciones leídas, al que se asigna el nombre leído de cada una como valor del atributo *nombre* de dicho objeto. Mientras que por otra parte, para cada condición creada, se realiza una consulta a la tabla *SentenciasCondiciones,* a través del script *ObtenerSentenciasCondicion.php.* Dicho script recibe como argumento el nombre de la condición que estamos tratando, y con su respuesta se inicializan las sentencias de dicho objeto PFCObjetoCondicion según sus respectivas sentencias (que obtiene de la tabla *SentenciasCondiciones)*.

Lectura de Servicios:

El acceso de lectura a la tabla *Servicios* se realiza durante la carga de la pantalla de configuración básica, es decir, en el método - *(void)viewDidLoad()* de su respectivo controlador de vista. Se sigue un esquema similar al de lecturas anteriores, para ello se llama a un método, ***leerTodosServicios(NSData *) datos,*** pasándole como argumentos los datos leídos de la tabla de datos S*ervicios* a través del script correspondiente. En este método se interpretan esos datos y se inicializan las variables que representarán dichos servicios dentro del sistema. A continuación se muestra un fragmento de código del método:

```
#define urlServicios [NSURL URLWithString: @"http://localhost/php/phpdao/generated/obtenerServicios.php"]

        ...
- (void) viewDidLoad {
        ...
    NSData *dataServicios = [NSData dataWithContentsOfURL:urlServicios];
    [self performSelectorOnMainThread:@selector(leerTodosServicios:) withObject:dataServicios waitUntilDone:YES];
}
```

```
- (void)leerTodosServicios:(NSData *)datos {
        ...
for (int j = 0; j < [serviciosLeidos count]; j++) {
    NSMutableDictionary *servicioActual = [[NSMutableDictionary alloc] initWithDictionary:[serviciosLeidos objectAtIndex:j]];
    NSMutableDictionary *perfilAsociado = [[NSMutableDictionary alloc] init];
    [self.todosServicios setObject:perfilAsociado forKey:[servicioActual objectForKey:@"nombre"]];
}

for (int i = 0; i < [serviciosLeidos count]; i++) {
    NSMutableDictionary *servicioActual = [[NSMutableDictionary alloc] initWithDictionary:[serviciosLeidos objectAtIndex:i]];
    NSLog(@"%@", servicioActual);
    // Si servicio tiene perfil asociado
    if ([servicioActual objectForKey:@"perfilAsociado"] != [NSNull null]) {
        NSString *perfilEntero = [servicioActual objectForKey:@"perfilAsociado"];
        [[self.todosServicios objectForKey:[servicioActual objectForKey:@"nombre"]] setObject:perfilEntero forKey:@"Perfil
            Actual"];
    }
}
self.nombresServicios = [[self.todosServicios allKeys] sortedArrayUsingSelector:@selector(compare:)];
```

Se hace un a petición a la URL del script (objeto *urlServicios*) y la respuesta se pasa como argumento al método *leerTodosServicios().* A raíz de este argumento este método va leyendo, como se puede observar en el primer bucle *for* del código, los nombres de los

servicios existentes, creando para almacenarlos un objeto **NSMutableDictionary** que representará a cada servicio leído. Este diccionario tendrá como claves los nombres de los respectivos servicios, y como objetos almacenados otros NSMutableDictionary que se inicializan en este mismo bucle, y contendrán el nombre (un NSString) del respectivo perfil asociado al servicio. Estos objetos creados se van almacenando en una variable de colección que guardará todos los servicios del sistema: ***self.todosServicios.*** En la segunda parte del método podemos ver como se comprueba, para cada servicio leído de la BD, si éste tiene un perfil asociado, en cuyo caso se inserta el nombre de dicho perfil en el diccionario que representa al perfil asociado dentro del servicio. Si el servicio leído no tiene perfil asociado (se lee NULL) dicho diccionario permanecerá vacío.

Lectura de Transiciones:

La lectura de las transiciones se realiza cada vez que aparece la vista '*Profiles Transitions',* para refrescar así los posibles cambios en las transiciones que haga el usuario a través de la interfaz. Para leer las transiciones definidas hará falta obtener por un lado las **condiciones** que provocarán las respectivas transiciones y por otro lado los **servicios** afectados en dichas transiciones. Para ello tendremos que leer las tablas ***Exchanges*** y ***ServiciosTransiciones*** respectivamente. Utilizando para ello llamadas a los scripts PHP correspondientes desde el código de los métodos que denotaremos a continuación:

- **Lectura de Condiciones que provocarán transiciones**

```
- (void)viewWillAppear:(BOOL)animated {
    NSURL *urlExchanges = [NSURL URLWithString: @"http://localhost/php/phpdao/generated/obtenerExchanges.php"];
    NSData *dataPerfiles = [NSData dataWithContentsOfURL: urlExchanges];
    [self performSelectorOnMainThread:@selector(leerExchangesServidor:) withObject:dataPerfiles waitUntilDone:YES];
        ...
```

```
- (void)leerExchangesServidor:(NSData *) datos{

NSArray* datosLeidos = [NSJSONSerialization JSONObjectWithData:datos options:kNilOptions error:&error];

        ...

for (int i = 0; i < [datosLeidos count]; i++) {
    NSMutableDictionary *datosActuales = [[NSMutableDictionary alloc] initWithDictionary:[datosLeidos objectAtIndex:i]];
    NSMutableDictionary *perfilFromActual = [self.todosPerfiles objectForKey:[datosActuales objectForKey:@"perfilFrom"]];
    NSMutableDictionary *condicionesPerfilFrom = [perfilFromActual objectForKey:@"keyCondiciones"];
    NSMutableDictionary *condicionesPerfilTo = [condicionesPerfilFrom objectForKey:[datosActuales objectForKey:@"perfilTo"]];

    [condicionesPerfilTo setObject:[self.todasCondiciones objectForKey:[datosActuales objectForKey:@"nombCondicion"]] forKey:
        [datosActuales objectForKey:@"nombCondicion"]];
```

El método de lectura de las condiciones sigue el mismo patrón que los anteriores: recibe un objeto NSData como argumento llamado *datos*, que contendrá todas las condiciones que intervengan en una transición en el sistema, es decir, el resultado de listar todas las filas

de la tabla ***Exchanges***. Éste método, para cada transición, obtiene el diccionario que representa el perfil From de la misma, y le añade una entrada consistente en otro diccionario, que contendrá como entradas pares clave:valor formados por el perfil destino y las condiciones que provocarán la transición a este perfil, respectivamente. Quedarán así preparados los datos que usará la lógica interna de la aplicación para mostrar la información en la interfaz gráfica de la aplicación.

- **Lectura de los servicios afectados en las transiciones**

```
- (void)leerServiciosFrom {
    NSError *error;
    NSString *getServiciosFrom = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated/
        obtener_servicios_transicion.php?from=%@&to=%@", [self.perfilFrom objectForKey:@"Perfil seleccionado"],[self.perfilTo
        objectForKey:@"Perfil seleccionado"]];
    NSURL *urlServiciosFrom = [NSURL URLWithString: getServiciosFrom];
    NSData *dataServiciosFrom = [NSData dataWithContentsOfURL: urlServiciosFrom];
    NSArray *serviciosFrom = [NSJSONSerialization JSONObjectWithData:dataServiciosFrom options:kNilOptions error:&error];
    self.serviciosPerfilFrom = [[NSMutableArray alloc] init];

    for (int i = 0; i < [serviciosFrom count]-1; i++) {
        NSMutableDictionary *servicioAsociadoExchangeActual = [[NSMutableDictionary alloc] initWithDictionary:[serviciosFrom
            objectAtIndex:i]];
        [self.serviciosPerfilFrom addObject:[servicioAsociadoExchangeActual objectForKey:@"servicio"]];
    }

        ...
```

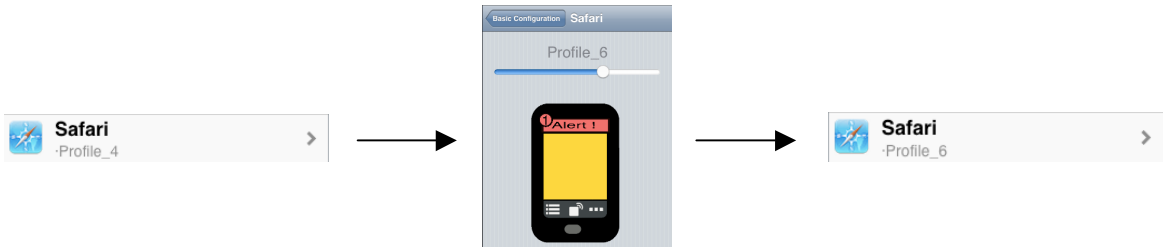
Éste método se encarga de leer los servicios que estén involucrados en una determinada transición de perfiles de molestia. No recibe ningún argumento ya que realiza la petición al script para leer la tabla ***ServiciosTransiciones*** desde el interior del propio método. Dicho script recibirá dos argumentos, los perfiles From y To involucrados en la transición, y responderá listando los servicios afectados en éstas (si existen). Los nombre de estos servicios serán almacenados en una variable de colección NSArray llamada *self.serviciosPerfilFrom*, tal y como se observa en la última parte del código.

Una vez llegados a este punto tenemos todos los objetos que la aplicación precisará leer de la base de datos preparados para poder mostrar correctamente la información en las diferentes pantallas que componen la interfaz gráfica, dejando así todo listo para que sea la capa de presentación quien lleve a cabo esta tarea.

En los siguientes apartados analizaremos todas las acciones que pueda realizar el usuario y que desencadenan la realización de alguna funcionalidad lógica de la aplicación que implique tener que volver a acceder a la capa de datos, bien sea para insertar, borrar o actualizar una fila de cualquiera de las tablas.

3.2.3.2 Asociando un perfil a un servicio

Cuando el usuario asocia un nuevo perfil a un servicio es necesario que este cambio se refleje en la capa de datos, en concreto habrá que actualizar el atributo *perfil_asociado* de una fila dentro de la tabla **Servicios**.



Para llevar a cabo esta labor, cada vez que se cambia el perfil de un servicio dentro del apartado de configuración básica de la app, se ejecuta el código que se muestra a continuación:

```
//Actualizamos Servicio Modificado
NSString *getInsertarServicio = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated/
    actualizar_servicio.php?servicio=%@&perfil=%@", [self.nombresServicios objectAtIndex:self.ultimaFila], perfilAsociado
    ];
NSURL *direccionInsertarServicio = [[NSURL alloc] initWithString:getInsertarServicio];
NSData *getDataInsertarServicio = [NSData dataWithContentsOfURL: direccionInsertarServicio];

        ...

//Borramos todas las serviciosTransiciones con el ultimo perfil asociado (en from) y el servicio principal
NSString *getBorrarServicioTransicion = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated/
    borrarServicioTransicionFromServicio.php?from=%@&servicio=%@", self.ultimoPerfilSeleccionado, [self.nombresServicios
    objectAtIndex:self.ultimaFila]];
NSURL *direccionBorrarServicioTransicion = [[NSURL alloc] initWithString:getBorrarServicioTransicion];
NSData *getDataInsertarServicioTransicion = [NSData dataWithContentsOfURL: direccionBorrarServicioTransicion];
```

Primero se actualiza la tupla de tabla ***Servicios*** correspondiente al servicio cuyo perfil ha variado. Para ello se utiliza el script *actualizar_servicio*, que recibe como argumentos el servicio a modificar y su nuevo perfil asociado. Después se borran todos los servicios de las transiciones afectadas por el servicio modificado, eliminando para ello filas de la tabla ***ServiciosTransiciones***. Al hacer esto, se observa que puede haber transiciones que hayan quedado sin servicio asociado (pero seguirán teniendo una o más condiciones asociadas), estas transiciones deberán ser eliminadas completamente del SGBD eliminando también las condiciones asociadas a ellas, para mantener así la coherencia de los datos. Podemos observar el código del script PHP que obtiene estas tuplas "huérfanas" para borrarlas de tabla *Exchanges* a continuación:

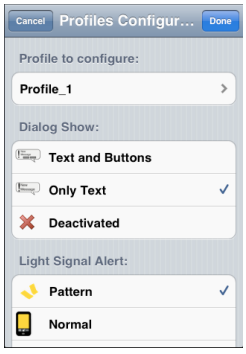
```
<?php
$con = mysql_connect("127.0.0.1","root","123456");
if (!$con)
{
  die('No se puede conectar a la BD Mysql: ' . mysql_error());
}
mysql_select_db("PFC", $con);
//Codificamos el resultado "manualmente" con JSON
$sql="SELECT DISTINCT perfil_from, perfil_to FROM Exchanges WHERE NOT EXISTS (SELECT * FROM ServiciosTransiciones WHERE
perfil_from = from_profile AND perfil_to = to_profile)";
$result=mysql_query($sql,$con);
echo '[';
while($row = mysql_fetch_array($result)){
  echo json_encode($row);
  echo ',';
  }
echo '{}]';
mysql_close($con);
?>
```

Este script obtiene los pares de perfiles From y To que se encuentren en la tabla *Exchanges* y no en la tabla *ServiciosTransiciones.* Su resultado será utilizado para borrar las condiciones asociadas a una transición que hayan quedado sin servicio afectado y por lo tanto sin transición, es decir, eliminaremos las tuplas que cumplan pertenecer a esa transición dentro de la tabla *Exchanges*. Se muestra el código que implementa lo explicado y se ejecutará justo después de eliminar los serviciosTransiciones:

```
//Obtenemos y borramos las transitions huerfanas, las que se han quedado sin servicioTransiciones
NSString *getObtenerTransicionesHuerfanas = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated/
    obtenerTransicionesHuerfanas.php"];
NSURL *direccionObtenerTransicionesHuerfanas = [[NSURL alloc] initWithString:getObtenerTransicionesHuerfanas];
NSData *getDataTransicionesHuerfanas = [NSData dataWithContentsOfURL: direccionObtenerTransicionesHuerfanas];
NSLog(@"%@", direccionObtenerTransicionesHuerfanas);
NSError *error;
NSArray *huerfanasLeidas = [NSJSONSerialization JSONObjectWithData:getDataTransicionesHuerfanas options:kNilOptions
    error:&error];
for (int j = 0; j < [huerfanasLeidas count]; j++) {
    NSMutableDictionary *huerfanaActual = [[NSMutableDictionary alloc] initWithDictionary:[huerfanasLeidas objectAtIndex:
        j]];
    NSString *getBorrarTransicionesHuerfanas = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated/
        borrar_todosExchanges.php?from=%@&to=%@", [huerfanaActual objectForKey:@"perfil_from"],[huerfanaActual
        objectForKey:@"perfil_to"]];
    NSURL *direccionBorrarTransicionesHuerfanas = [[NSURL alloc] initWithString:getBorrarTransicionesHuerfanas];
    NSData *getDataBorrarTransicionesHuerfanas = [NSData dataWithContentsOfURL:direccionBorrarTransicionesHuerfanas];

    ...
```

Primero se llama al script comentado anteriormente y usamos su resultado para borrar las tuplas de la tabla *Exchanges* cuyos perfiles From y To coincidan con él, utilizando para ello el script de borrado que se aprecia en el bucle for.

3.2.3.3 Modificando el grado de intromisión de un perfil



Cuando el usuario modifica algún perfil variando el nivel de molestia de algún mecanismo de interacción al pulsar una fila de la tabla, automáticamente aparece un icono **"Done"** en la parte derecha de la barra superior, mientras que en la parte izquierda aparecerá otro botón **"Cancel"**. Cuando el usuario pulsa el botón Done, los cambios se guardarán en la capa de datos, mientras que si éste decide cancelar, los cambios serán ignorados y se volverán a reiniciar los "ticks" de las filas conforme al perfil original. Ambos botones se han creado mediante el siguiente código:

```
UIBarButtonItem *botonAceptar = [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemDone target:self action:@selector
    (guardarCambios:)];
self.navigationItem.rightBarButtonItem = botonAceptar;
```

```
UIBarButtonItem *botonCancelar = [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemCancel target:self action:@selector
    (cancelarCambios:)];
```

action representa el método que lanzarán al ser pulsados, mientras que **target** hace referencia a la clase donde están implementados dichos métodos, en este caso **self,** es decir, ella misma.

gura 29. Creación de transición añadiendo una condición

Estos botones lanzarán dos acciones (**IBAction**), *guardarCambios* y *cancelarCambios*, analizaremos la utilidad de estos métodos a continuación:

- **guardarCambios:**

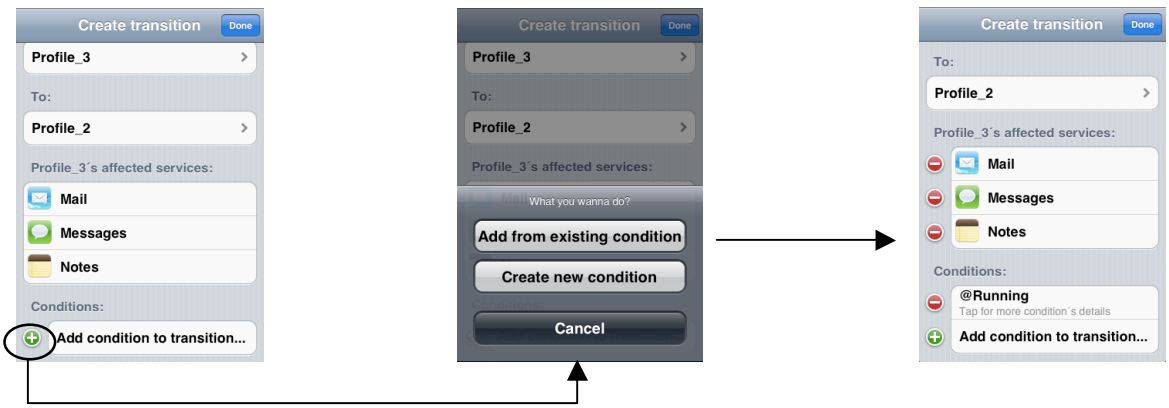
```
- (IBAction) guardarCambios:(id)sender {

    NSMutableDictionary *copiaDeTemporal = [[NSMutableDictionary alloc] initWithDictionary:self.perfilModificadoTemporal
        copyItems:YES];
    [self.todosPerfiles removeObjectForKey:[self.nombrePerfil objectForKey:@"Perfil seleccionado"]];
    [self.todosPerfiles setObject:copiaDeTemporal forKey:[self.nombrePerfil objectForKey:@"Perfil seleccionado"]];
    NSString *getModificarPerfil = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated/actualizar_perfil.php
        ?sound=%@&speech=%@&vibration=%@&light=%@&statusBar=%@&toast=%@&dialog=%@&nombre=%@", [copiaDeTemporal objectForKey:
        @"keySound"], [copiaDeTemporal objectForKey:@"keySpeech"],[copiaDeTemporal objectForKey:@"keyVibration"],[copiaDeTemporal
        objectForKey:@"keyLightSignal"],[copiaDeTemporal objectForKey:@"keyStatusBar"],[copiaDeTemporal objectForKey:@"keyToast"]
        ,[copiaDeTemporal objectForKey:@"keyDialog"],[self.nombrePerfil objectForKey:@"Perfil seleccionado"]];
    NSURL *direccionModificarPerfil = [[NSURL alloc] initWithString:getModificarPerfil];
    NSData *getDataModificarPerfil = [NSData dataWithContentsOfURL: direccionModificarPerfil];

        ...
```

Este método se ejecutará cuando el usuario presioné el botón **Done**, y como podemos observar actualiza un perfil (fila) de la tabla *Perfiles* del sistema de base de datos. Utiliza para ello un script al que se le pasa tanto todos los nuevos valores de los diferentes mecanismos de alerta definidos por el usuario, como el nombre del perfil a modificar. Todo ello será enviado como argumentos dentro de la URL de la petición HTTP.

- **cancelarCambios:** Este método se lanzará al pulsar el botón **Cancelar,** y simplemente desecha el perfil temporal en el que se almacenaban los cambios dentro de la aplicación. Además realiza toda la lógica necesaria para volver a mostrar los "ticks" de las filas conforme al perfil original seleccionado, recargando el contenido de la vista (y sus tablas) mediante el método: [self.tableView reloadData];

3.2.3.4 Creando una transición

Las transiciones son creadas en la vista de nombre "Edit Transitions", para ello el usuario debe seleccionar un *Perfil From* que tenga asociado algún servicio y un perfil To al que se cambiará. Después de ello deberá **añadir una condición** a la transición para crearla. La condición añadida puede ser una condición existente o una creada en ese mismo momento, la principal diferencia entre ambos casos, es que si el usuario decide crear una condición, ésta será añadida la sistema (a la tabla ***condiciones*** y ***SentenciasCondiciones***) antes de guardar la transición propiamente dicha, es decir antes de guardar la condición que la origina en la tabla *Exchanges,* y los servicios afectados en la tabla *ServiciosTransiciones*.



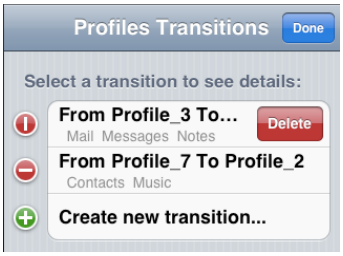
Una vez creada la transición, ésta se podrá editar pudiendo agregar o eliminar más condiciones a la transición, o eligiendo qué servicios ser verán afectados en el cambio de perfil de molestia. Analizaremos el código que permite la creación de una transición, en primer lugar se comprueba si la condición añadida (bien creada o existente) crea una transición, en cuyo caso se obtienen todos los servicios asociados al perfil From y se insertan en la tabla serviciosTransiciones, ya que ahora estarán envueltos en una transición. Se muestra el código que implementa este comportamiento a continuación:

```
if(self.anyadiendoCondicion == YES || self.creandoCondicion == YES){

    // Si añadimos una condicion que crea un exchange, metemos todos los servicios del perfilFrom en serviciosTransiciones
    NSError *error;
    NSString *getServiciosAsociadosFrom = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated/
        obtener_servicios_transicion.php?from=%@&to=%@", [self.perfilFrom objectForKey:@"Perfil seleccionado"],[self.perfilTo
        objectForKey:@"Perfil seleccionado"]];
    NSURL *urlServiciosAsociadosFrom = [NSURL URLWithString: getServiciosAsociadosFrom];
    NSData *dataServiciosAsociadosFrom = [NSData dataWithContentsOfURL: urlServiciosAsociadosFrom];
    NSArray *serviciosAsociadosFrom = [NSJSONSerialization JSONObjectWithData:dataServiciosAsociadosFrom options:kNilOptions
        error:&error];

    //No habia condicion para este exchange, metemos todos los servicios de to en serviciosTransiciones
    if ([[serviciosAsociadosFrom objectAtIndex:0] count] == 0) {
        self.sinCondicion = NO;
        NSError *error2;
        NSString *getServiciosFrom = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated/
            obtenerServiciosPerfilAsociado.php?perfilAsociado=%@", [self.perfilFrom objectForKey:@"Perfil seleccionado"]];
        NSURL *urlServiciosFrom = [NSURL URLWithString: getServiciosFrom];
        NSData *dataServiciosFrom = [NSData dataWithContentsOfURL: urlServiciosFrom];
        NSArray *serviciosFrom = [NSJSONSerialization JSONObjectWithData:dataServiciosFrom options:kNilOptions error:&error2]
            ;

        for (int i = 0; i < [serviciosFrom count]; i++) {
            NSMutableDictionary *servicioAsociadoExchangeActual = [[NSMutableDictionary alloc] initWithDictionary:
                [serviciosFrom objectAtIndex:i]];
            //Por servicio cuyo perfil asociado sea from, lo metemos en tablas serviciosTransitciones
            NSString *getServiciosFrom = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated/
                insertar_servicio_transicion.php?from=%@&to=%@&servicio=%@", [self.perfilFrom objectForKey:@"Perfil
                seleccionado"], [self.perfilTo objectForKey:@"Perfil seleccionado"], [servicioAsociadoExchangeActual
                objectForKey:@"nombre" ]];
            NSURL *urlServiciosFrom = [NSURL URLWithString: getServiciosFrom];

            NSData *dataServiciosFrom = [NSData dataWithContentsOfURL: urlServiciosFrom];
            NSLog(@"%@", dataServiciosFrom);
        }

    }
}



        …
```

El código se divide en tres partes, en la primera de ellas se comprueba si la condición añadida (creada o existente) crea una transición entre los perfiles seleccionados, utilizando para ello una llamada al script *obtener_servicios_transicion.php* que recibe como argumento el perfil From y To de la transición cuyos servicios queremos obtener. Si este script no devuelve ningún valor, condición que se comprueba en el segundo *if*, significará que no existía previamente esa transición y por lo tanto pasaremos a crearla en la siguiente parte del código. En la segunda parte, se obtienen todos los servicios que tenga asociados el perfil From seleccionado, usando el script *obtenerServiciosPerfilAsociado.php.* Por último en la última parte del código, la correspondiente al bucle *for*, se insertan todos esos servicios del perfil afectados en la transición en la tabla ***serviciosTransiciones.*** Una vez tenemos añadidos todos los servicios afectados en la transición, pasaremos a añadir la respectiva condición que lanza la transición en la tabla ***Exchanges,*** que recordemos contiene todas las condiciones afectadas en una transición. Esto se realizará mediante una llamada al script *insertar_exchange.php,* que recibe como argumentos el perfil From, perfil To y nombre de la condición a añadir tal y como se muestra a continuación:

```
if (anyadiendoCondicion == YES) {

    //Añadimos el nuevo Exchange creado a la BD
    NSString *getInsertarExchange = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated/
        insertar_exchange.php?from=%@&to=%@&nomCond=%@", [self.perfilFrom objectForKey:@"Perfil seleccionado"], [self.
        perfilTo objectForKey:@"Perfil seleccionado"], [self.condicionAnyadida objectForKey:@"Perfil seleccionado"]];
    NSLog(@"%@",getInsertarExchange);

    NSURL *direccionInsertarExchange = [[NSURL alloc] initWithString:getInsertarExchange];

    NSData *getDataInsertarExchange = [NSData dataWithContentsOfURL: direccionInsertarExchange];
    NSLog(@"%@", getDataInsertarExchange);

    self.anyadiendoCondicion = NO;
}
```

3.2.3.5 Borrando una transición



El usuario podrá eliminar una transición que ya no le vaya a ser de utilidad. El mecanismo para confirmar la eliminación lo ofrece el SDK de iOS, al requerir tocar dos veces, una el símbolo (-) y otra el botón *Delete*, para eliminar una fila de la tabla. Al eliminar una transición, deberemos eliminar todos los servicios afectados en las transiciones así como las condiciones que la provocaban. Esto se traducirá en nuestro sistema de datos debiendo borrar las tuplas de las tablas ***Exchanges*** y ***ServiciosTransiciones*** correspondientes a la transición a eliminar, para reflejar así este cambio en la capa de datos.

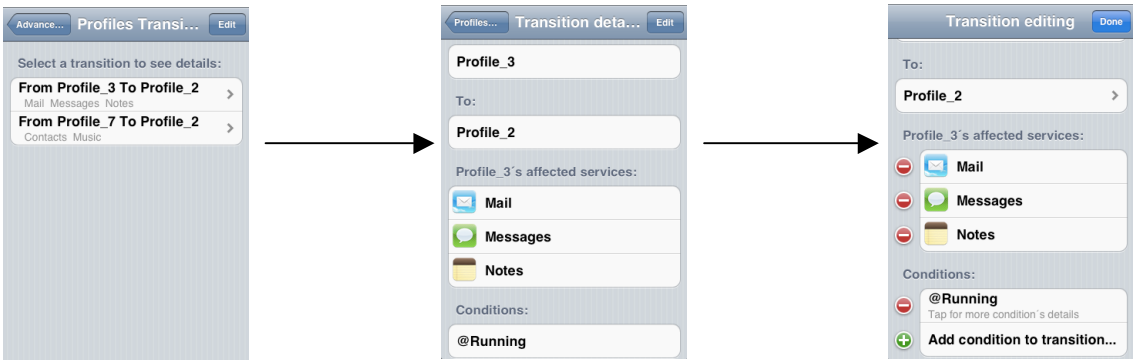
Se muestra el código que lleva a cabo estas dos funciones de borrado en la captura de la parte inferior. Se puede observar como en primer lugar se ejecuta un script que borra todas las filas de la tabla ***Exchanges*** con un perfil From y to determinados, que se les pasan como argumentos al script. Con ello habremos eliminado del sistema las condiciones que lanzaban esa transición (en la tabla *Exchanges* NO tabla *Condiciones*). En segundo lugar se eliminan las filas de la tabla ***ServiciosTransiciones*** correspondientes a dicha transición, para ello se llama al script *borrarServicioTransicioFromTo.php* que elimina las tuplas de esta tabla cuyos perfiles From y to coincidan con los pasados como argumentos en la URL del script. Habiendo borrado tras su ejecución todos los servicios asociados a dicha transición. Una vez se han realizado todos estos pasos, la transición quedará totalmente eliminada del sistema:

```
- (void)tableView:(UITableView *)TableView commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath {


//Borramos todos los exchanges (transitions) de la base de datos
NSString *getTodosExchagesBorrado = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated/
    borrar_todosExchanges.php?from=%@&to=%@", nombreFrom, nombreTo];
NSURL *direccionExchagesBorrado = [[NSURL alloc] initWithString:getTodosExchagesBorrado];
NSData *getDataExchagesBorrado = [NSData dataWithContentsOfURL: direccionExchagesBorrado];
NSLog(@"%@", getDataExchagesBorrado);

//Borramos Todos los serviciosTransiciones (from y to) se la base de datos
NSString *getBorrarServicioTransition = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated/
    borrarServicioTransicionFromTo.php?from=%@&to=%@", nombreFrom,nombreTo];

NSURL *direccionBorrarServicioTransition = [[NSURL alloc] initWithString:getBorrarServicioTransition];
NSLog(@"%@", direccionBorrarServicioTransition);

NSData *getDataBorrarServicioTransition = [NSData dataWithContentsOfURL: direccionBorrarServicioTransition];
NSLog(@"%@", getDataBorrarServicioTransition);


        ...
```

Este código se encuentra dentro del método ***commitEditingStyle***, que se ejecutará al borrar/añadir una fila en la tabla. Nótese que la clase que implementa esta vista puede utilizar el método gracias a descender (heredar) de la clase ***UITableView.***

3.2.3.6 Editando una transición

Una vez se ha creado una transición el usuario podrá editarla. Para ello podrá elegir los servicios afectados en ella y añadir o borrar condiciones que provoquen dicha transición. Para esto deberá seleccionar en primer lugar la transición a editar, tras esto se abrirá una ventana que mostrará la información de la transición que queremos modificar, pero no se podrá editar. Entonces deberá pulsar el botón *Edit* y como resultado la transición se volverá editable, tras realizar todos los cambios oportunos el usuario deberá pulsar el botón *Done,* estos se guardarán de manera persistente y la tabla volverá a modo no editable.



- **Eliminación de servicios de una transición**

Se podrán decidir qué servicios de los asociados al perfil From se verán afectados en transición a tratar, para ello podremos eliminar las filas de los servicios que no queramos que intervengan en ella. Cada fila borrada de servicio repercutirá en los datos como una fila borrada dentro de la tabla ***ServiciosTransiciones.*** Si el usuario elimina todos los servicios de una transición, ésta deberá desaparecer, por lo que habrá que borrar también todas las tuplas de la tabla ***Exchanges*** que representen a las condiciones que lancen la respectiva transición. Se muestra el código, implementado dentro del método que gestiona el borrado de filas de una tabla, que ejecuta dichas acciones a continuación:

```
- (void)tableView:(UITableView *)TableView commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath {

                ...

else {      //Seccion de servicios del from
    //Borramos servicio transition de BD
    NSString *getBorrarServicioTransition = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated/
        borrarServicioTransicion.php?from=%@&to=%@&servicio=%@", [self.perfilFrom objectForKey:@"Perfil seleccionado"], [self
        .perfilTo objectForKey:@"Perfil seleccionado"], [self.serviciosPerfilFrom objectAtIndex:[indexPath row]]];
    NSURL *direccionBorrarServicioTransition = [[NSURL alloc] initWithString:getBorrarServicioTransition];
    NSData *getDataBorrarServicioTransition = [NSData dataWithContentsOfURL: direccionBorrarServicioTransition];

    // Si tras borrar el servicio, no hay mas serviciosTransiciones, eliminamos la transicion from y to (eliminar todas las
        condiciones)
    NSError *error;
    NSString *getServiciosAsociadosFrom = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated/
        obtener_servicios_transicion.php?from=%@&to=%@", [self.perfilFrom objectForKey:@"Perfil seleccionado"],[self.perfilTo
        objectForKey:@"Perfil seleccionado"]];
    NSURL *urlServiciosAsociadosFrom = [NSURL URLWithString: getServiciosAsociadosFrom];
    NSData *dataServiciosAsociadosFrom = [NSData dataWithContentsOfURL: urlServiciosAsociadosFrom];
    NSArray *serviciosAsociadosFrom = [NSJSONSerialization JSONObjectWithData:dataServiciosAsociadosFrom options:kNilOptions
        error:&error];

    if ([[serviciosAsociadosFrom objectAtIndex:0] count] == 0) {

      //Guardamos borrado de exchanges en la BD
      NSString *getTodosExchagesBorrado = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated/
          borrar_todosExchanges.php?from=%@&to=%@", [self.perfilFrom objectForKey:@"Perfil seleccionado"], [self.perfilTo
          objectForKey:@"Perfil seleccionado"]];
      NSURL *direccionExchagesBorrado = [[NSURL alloc] initWithString:getTodosExchagesBorrado];
      NSData *getDataExchagesBorrado = [NSData dataWithContentsOfURL: direccionExchagesBorrado];
      NSLog(@"%@", getDataExchagesBorrado);
      [self.tableView reloadData];


                ...
```

La funcionalidad del código es la siguiente: primero se elimina del sistema el servicio suprimido de la transición, borrando para ello la fila correspondiente de la tabla ***ServiciosTransiciones*****,** mediante el uso del script *borrarServicioTranscion.php* que recibe los perfiles From y To de la transición y el nombre del servicio que queremos eliminar. Una vez borrado el servicio, se comprueba si ése era el último servicio asociado a la transición, utilizando para ello el script *obtener_servicios_transiciones.php* pasándole como argumentos los respectivos perfiles afectados en la transición, y si este script no devuelve ningún servicio (condición que se comprueba en la cláusula *if*) deberemos asimilar que la transición se ha quedado sin servicios, y por lo tanto pasaremos a eliminarla completamente del sistema. Para lograr esto borraremos todas las tuplas de esa transición que hayan quedado en la tabla ***Exchanges*** mediante el script *borrar_todosExchanges.php*. Este script recibe los perfiles From y To de los cuales se borrarán todas las condiciones (filas de *Exchanges*) que lanzaban la transición. Tras esta secuencia quedará el servicio borrado y la consistencia de la capa de datos garantizada.

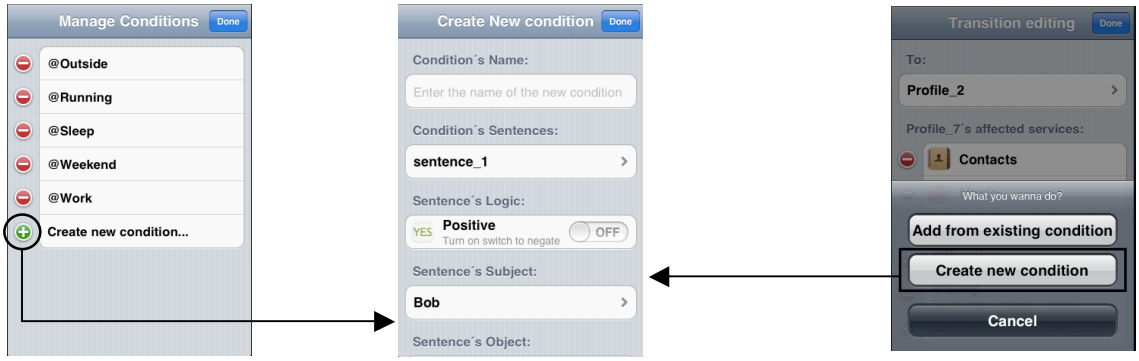
- **Eliminación de condiciones de una transición**

La eliminación de condiciones, se realiza desde el mismo método (ya que son filas borradas de la misma tabla) que la de servicios, y sigue también la misma metodología. En primer lugar se borra la condición de transición que el usuario ha eliminado de la base de datos ***Exchanges***, y si ésta condición era la última que tenía asociada dicha transición, se pasa a borrar todas las tuplas de la tabla ***ServiciosTransiciones*** de la tabla de datos pertenecientes a dicha transición eliminando así todo rastro de ella. Se puede observar que es un comportamiento idéntico al expuesto con el borrado de servicios, solo que cambiando el

orden de las tablas a las que se accede. Se muestra el código que realiza esta tarea a continuación:

```
- (void)tableView:(UITableView *)TableView commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath {

                ...

//Borramos exchange de BD
NSString *getBorrarC = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated/borrar_exchange.php?
    from=%@&to=%@&nomCond=%@", [self.perfilFrom objectForKey:@"Perfil seleccionado"], [self.perfilTo objectForKey:
    @"Perfil seleccionado"], [arrayNombresCondiciones objectAtIndex:fila]];
NSURL *direccionBorrarC = [[NSURL alloc] initWithString:getBorrarC];
NSData *getDataBorrarC = [NSData dataWithContentsOfURL: direccionBorrarC];
NSLog(@"%@", getDataBorrarC);
 [self.tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath] withRowAnimation:
     UITableViewRowAnimationAutomatic];

//Borramos todos serviceTransition en BD con from y to, si era la última condicion (si desaparece la exchange)
if ([condicionesPerfilTo count] == 0){

    NSString *getBorrarServicioTransition0 = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated
        /borrarServicioTransicionFromTo.php?from=%@&to=%@", [self.perfilFrom objectForKey:@"Perfil seleccionado"],
        [self.perfilTo objectForKey:@"Perfil seleccionado"]];
    NSLog(@"%@", getBorrarServicioTransition0);

    NSURL *direccionBorrarServicioTransition0 = [[NSURL alloc] initWithString:getBorrarServicioTransition0];
    NSData *getDataBorrarServicioTransition0 = [NSData dataWithContentsOfURL: direccionBorrarServicioTransition0];

                ...
```

3.2.3.7 Creando una condición

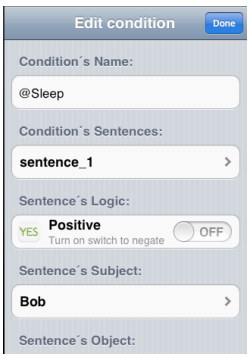


Al insertar el usuario una condición al sistema por medio de la interfaz, ya sea en la pantalla de editar condiciones o al añadir una condición a una transición, se realizarán sobre la capa de datos los mismos cambios. Estos cambios consistirán en primer lugar en añadir la condición creada a la tabla ***Condiciones***, y después insertar sus respectivas sentencias a la tabla ***SentenciasCondiciones.*** Se muestra el código que implementa este comportamiento, que se ejecutará al pulsar el botón *Done,* de la barra superior:

```
//Actualizamos BD, primero introducimos condicion, y luego todos sentenciasCondiciones
NSString *getInsertar = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated/insertar_condicion.php?
    nombre=%@", self.condicionCreada.nombreCondicion];
NSURL *direccionInsertar = [[NSURL alloc] initWithString:getInsertar];
NSData *getDataInsertar = [NSData dataWithContentsOfURL: direccionInsertar];
NSLog(@"%@", getDataInsertar);
//Tras meter el nombre de la condicion, metemos sentenciasCondiciones
NSMutableDictionary *sentencesActuales = [[NSMutableDictionary alloc] initWithDictionary:self.condicionCreada.sentences
    copyItems:YES];
NSArray *nombresTodasSentences = [[sentencesActuales allKeys] sortedArrayUsingSelector:@selector(compare:)];
for (int i = 0; i < [sentencesActuales count]; i++) {
    NSMutableDictionary *sentenceActual = [[NSMutableDictionary alloc] initWithDictionary:[sentencesActuales
        objectForKey:[nombresTodasSentences objectAtIndex:i]]];
    NSString *getInsertarSentencias = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated/
        insertarSentenciaCondicion?nombre=%@&subject=%@&object=%@&predicats=%@&negada=%@", self.condicionCreada.
        nombreCondicion, [sentenceActual objectForKey:@"subject"],[sentenceActual objectForKey:@"object"],[sentenceActual
        objectForKey:@"predicats"],[sentenceActual objectForKey:@"negada"]];
    NSURL *direccionInsertarSentencias = [[NSURL alloc] initWithString:getInsertarSentencias];
    NSData *getDataInsertarSentencias = [NSData dataWithContentsOfURL: direccionInsertarSentencias];
    NSLog(@"%@", getDataInsertarSentencias);
self.creandoCondicion = NO;
}
```

Se observa que se inserta la condición en la tabla ***Condiciones*** utilizando el script *insertar_condicion.php* al que se le pasa el nombre de condición a insertar como argumento. Una vez insertada la condición, pasaremos a la inserción de sus respectivas sentencias usando para tal fin el script *insertarSentenciaCondicion.php* que recibe como argumentos los respectivos, subject, object y Predicat y la lógica de cada sentencia que formará la condición. Se ejecutará este script tanta veces como sentencias contenga la condición (bucle *for*), y en cada llamada se añadirá una fila en la tabla ***SentenciasCondiciones*** de la capa de datos.

3.2.3.8 Editando una condición

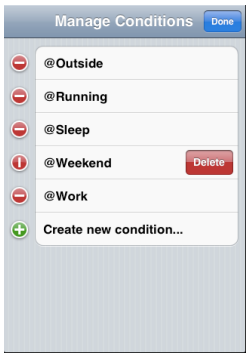


Durante la edición de una condición, el usuario podrá tanto cambiar el nombre de la condición, como crear nuevas sentencias o editar las existentes de la condición que se esté tratando en ese momento. La metodología que sigue la lógica de la capa de negocio de la aplicación para ello consiste en **actualizar** el nombre de la condición modificada por el usuario en la tabla condiciones de la BD. Nótese que los cambios se transmitirán en cascada. Se muestra la llamada al script más abajo:

```
...

if (self.modificandoCondicion == YES) {

    NSString *getActulizar = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated/actualizar_condicion.
        php?nombre=%@&nombreOrig=%@", self.condicionOriginal.nombreCondicion, self.nombreClaveAnterior];
    NSURL *direccionActualizar = [[NSURL alloc] initWithString:getActulizar];
    NSData *getDataActualizar = [NSData dataWithContentsOfURL: direccionActualizar];
    NSLog(@"%@", getDataActualizar);

    //Borramos todas las sentenciasCondiciones anteriores de esa condicion, para despues meter las nuevas modificadas.
    NSString *getBorrarSentencias = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated/
        borrarSentenciasCondicion.php?nombre=%@", self.condicionOriginal.nombreCondicion];
    NSURL *direccionBorrarSentencias = [[NSURL alloc] initWithString:getBorrarSentencias];
    NSData *getDataBorrarSentencias = [NSData dataWithContentsOfURL: direccionBorrarSentencias];

...
```

Se puede observar como se actualiza el nombre de la condición en la tabla ***Condiciones***, este cambio repercutirá en la tabla ***Exchanges*** y ***SentenciasCondiciones*** gracias a la restricción de integridad de clave ajena con dichas tablas. Para llevar a cabo esta acción se llama al script *actualizar_condicion.php* que recibe como argumentos el nombre de la condición a modificar y el nuevo nombre. Se puede observar que después se borran las sentencias de la condición mediante el script *borrarSentenciasCondicion.php,* para posteriormente introducir las nuevas modificadas por el usuario en la tabla ***SentenciasCondiciones*** de la base de datos del sistema, de la misma manera que se introducían en la creación de condiciones del apartado anterior.

3.2.3.9 Borrando una condición



Cuando el usuario elimina una condición, estos cambios se verán repercutidos inmediatamente en la tabla ***Condiciones*** de base de datos. Al eliminar una fila de ésta tabla, se borrarán en cascada las filas con atributos con restricción de clave ajena de las tablas ***SentenciasCondiciones***, que representa, como ya hemos hablado, las sentencias asociadas a una condición; y de la tabla ***Exchanges,*** que contiene las condiciones afectadas en una transición. Gracias a este comportamiento propio del lenguaje de programación SQL, logramos mantener la integridad los datos que necesita la aplicación. A continuación se muestra el código que borra la condición de la base de datos:

```
- (void)tableView:(UITableView *)TableView commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath {

        ...

   if (editingStyle == UITableViewCellEditingStyleDelete) {
       //Manejador de Eliminacion
       NSString *get = [[NSString alloc] initWithFormat:@"http://localhost/php/phpdao/generated/borrar_condicion.php?nombre=%@",
           [arrayNombresCondiciones objectAtIndex:fila]];
       [self.todasCondiciones removeObjectForKey:[arrayNombresCondiciones objectAtIndex:fila]];
       NSURL *direccion = [[NSURL alloc] initWithString:get];
       NSData *getData = [NSData dataWithContentsOfURL: direccion];
       NSLog(@"%@", getData);
       [self.tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath] withRowAnimation:
           UITableViewRowAnimationAutomatic];

   } else if (editingStyle == UITableViewCellEditingStyleInsert) {
       //Manejador de Inserción


        ...
```

4 Escenarios de uso

En este último capítulo mostraremos una serie de escenarios reales de uso de la aplicación con los que se pretende analizar de manera práctica la utilidad que puede aportar a un determinado perfil de usuario en las diferentes tareas que realiza en su vida cotidiana. Tomaremos el papel de Bob, un profesor de Universidad que desea administrar la manera en que envían las notificaciones los servicios que utiliza habitualmente, según el contexto en que se encuentra o la prioridad que Bob dé a unos servicios o a otros. Para que el usuario pueda ajustar las diferentes notificaciones, hemos agregado los servicios (y sus respectivos iconos), así como los subjects, objects y predicats que éste utiliza en la capa de datos en el SGBD del servidor. De esta manera, Bob podrá ajustar los diferentes perfiles para los servicios que utiliza y crear sus respectivas transiciones según el contexto en que se encuentra (condiciones), de manera sencilla mediante una interfaz limpia e intuitiva.

Dividiremos este tema en tantos apartados como servicios utilice este usuario. Analizaremos en cada uno de ellos los perfiles que controlan las notificaciones que envía cada servicio y las transiciones entre éstos que se llevarán a cabo según el contexto en que se encuentre el profesor Bob, siendo comprobado según las condiciones que éste haya definido. Finalmente mostraremos como se presentan éstas notificaciones en el teléfono.

4.1 WashingMachine

Este servicio recibe las notificaciones que envía la lavadora de última generación que tiene Bob en su casa. Avisando así al usuario en su teléfono cuando se encuentra llena de carga o cuando el proceso de lavado haya terminado.

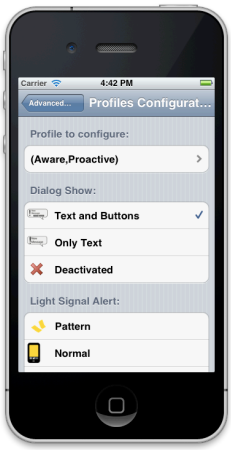
Descripción escenario:

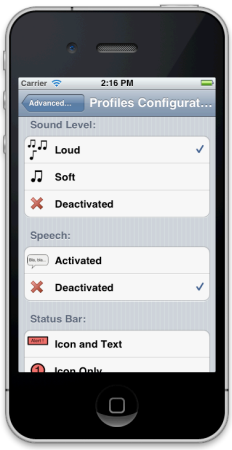
Bob desea que este servicio le avise de manera que sea **totalmente consciente** de la notificación enviada, a no ser que **se encuentre en una habitación diferente de donde esté el dispositivo en ese momento**, en cuyo caso la notificación tendrá que tener **un nivel de intromisión menor** al no poder ser atendida de inmediato. Así Bob podrá ser notificado de manera menos intrusiva cuando por ejemplo esté en la ducha y se le notifique que la lavadora está llena mientras el móvil recarga la batería en el dormitorio de Bob; o por otro parte será notificado de manera notoria mientras ve la televisión solo en su casa.

Creación de Perfiles:

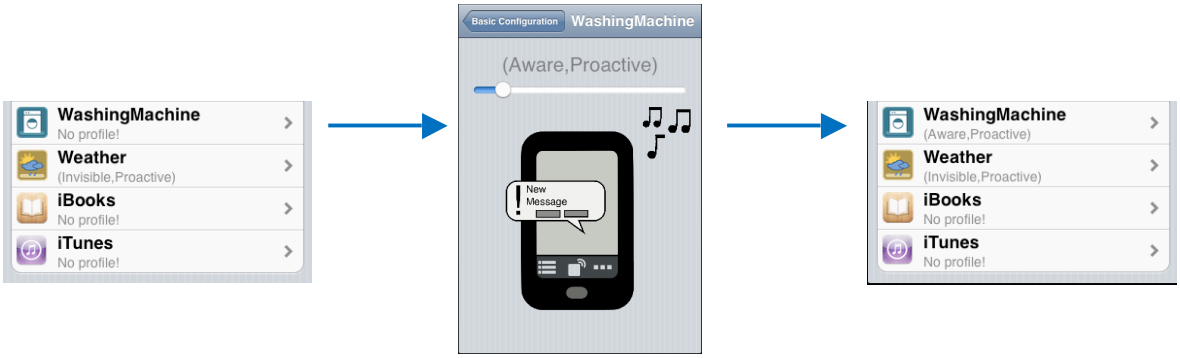
Necesitaremos crear dos perfiles, uno de ellos con intrusión baja (*slightly*) y otro que alerte al usuario de manera sonora y visual, con una intrusión más alta (*aware*). Analizaremos la creación de cada uno de ellos a continuación:

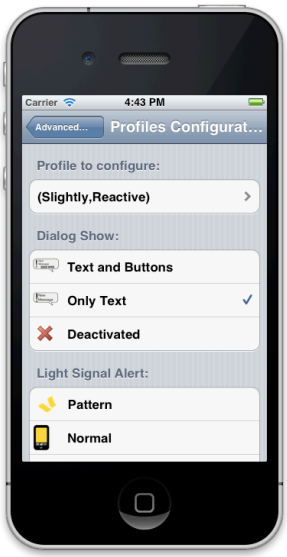
Figura 31. Asociación de perfil por defecto para servicio **WashingMachine**





Este perfil **(Aware,Proactive)** será el perfil asociado por defecto al servicio, y como podemos observar en la imagen de la izquierda, avisa al usuario mostrando un mensaje compuesto por texto y botones y emitiendo además un sonido con volumen alto y larga duración. Una vez que el usuario haya configurado el perfil, deberá asociarlo al servicio deseado, en este caso **WashingMachine**, utilizando la pantalla que se ofrece para ello dentro del apartado "*Basic Configuration*". Este proceso se muestra en las capturas de la parte inferior:



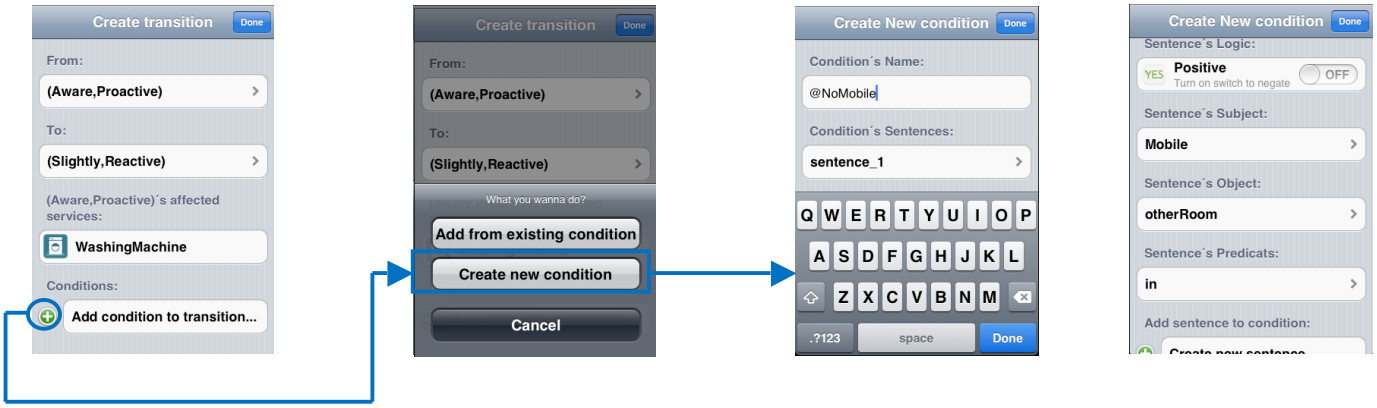


Este segundo perfil **(Slightly,Reactive)** es el elegido para crear el perfil con un nivel de intrusividad menor, al que se cambiará al producirse la transición. Como se puede ver, su único mecanismo de alerta asociado será la impresión en pantalla de un mensaje (sin botones). Este mecanismo tiene el fin de pasar desapercibido en un primer momento ya que no se le puede dar una atención inmediata al encontrarse el usuario en una ubicación diferente a la del dispositivo.

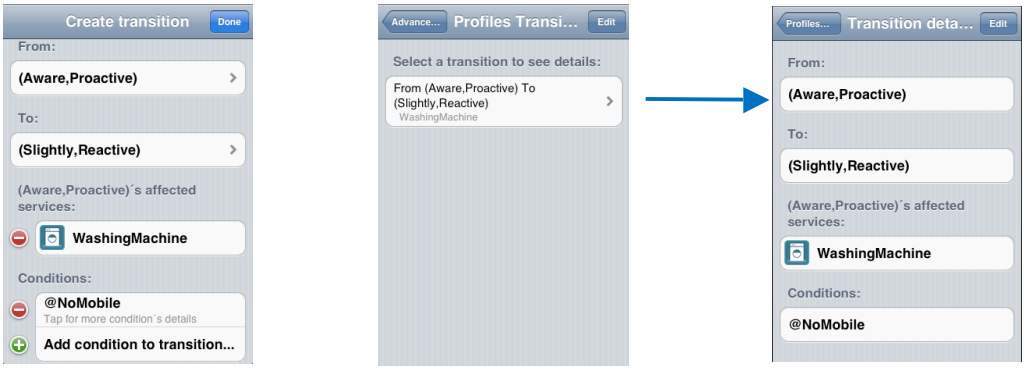
Figura 32. Creación de transición para servicio **WashingMachine**

Creación de Transicion:

Cuando Bob se encuentre en una habitación diferente a la del teléfono, si llega una notificación del servicio *WashingMachine,* deberá ser emitida con un nivel de alerta menor. Para generar este comportamiento se deberá crear una transición para ésta condición, tal y como se muestra en las siguientes capturas:

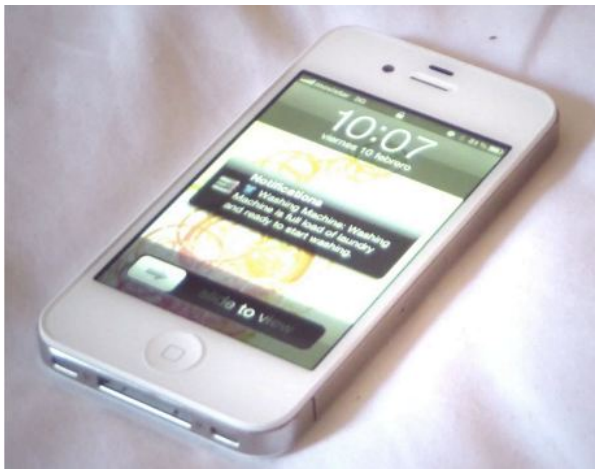


Primero se selecciona el perfil From, que será el perfil asociado por defecto al servicio, y después se debe seleccionar el Perfil To al que se cambiará durante la transición, en este caso el **(Aware,Proactive)** y **(Slightly,Reactive)** respectivamente. Una vez seleccionados los perfiles pasamos a crear la condición. Se puede observar que se crea una condición llama *@NoMobile* que cuenta con una sentencia, ***Mobile in otherRoom,*** que causará el comportamiento deseado por Bob. Una vez creada la condición, se pulsa el botón Done de ambas pantallas (*Create Condition* y *Create Transition*) y la transición quedará creada tal y como se muestra en las capturas de la parte inferior:



Mostrado de notificaciones:

A continuación se muestran dos capturas de como llegarían las notificaciones al dispositivo antes de la transición: mostrando **un mensaje con texto descriptivo** de la notificación emitida con botones con sus respectivas **acciones** y también mediante **un sonido con volumen alto (Aware,Proactive)**; y después de la transición: alertando únicamente **con un mensaje en pantalla (Slightly,Reactive)**. Imágenes de la izquierda y derecha respectivamente:



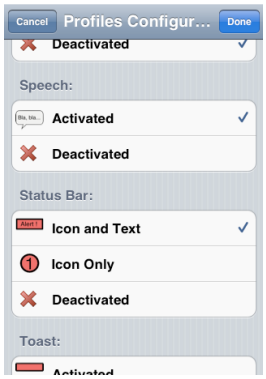
4.2 HealthCare

Esta aplicación envía recordatorios al usuario sobre los medicamentos que debe tomar.

Descripción escenario:

A Bob le interesa que esta aplicación le avise por defecto de manera **sonora y notoria** (**aware**), ya que es importante para su salud cumplir estos horarios. Se pretende que así se avise salvo cuando **esté comiendo en compañía**, en cuyo caso se le avisará de una manera más discreta (**slightly**).

Creación de Perfiles:



El perfil que se asociará por defecto a este servicio, **(Aware-Speech,Proactive)**, enviará las notificaciones mostrando un mensaje en que se imprimirá en la parte superior de la pantalla con el icono de la aplicación y un texto explicativo sobre ella. Además se le avisará de manera sonora mediante *Speech* al usuario gracias a los mecanismos de accesibilidad que ofrece iOS.

Figura 33. Asociación de perfil por defecto para servicio **HealthCare**

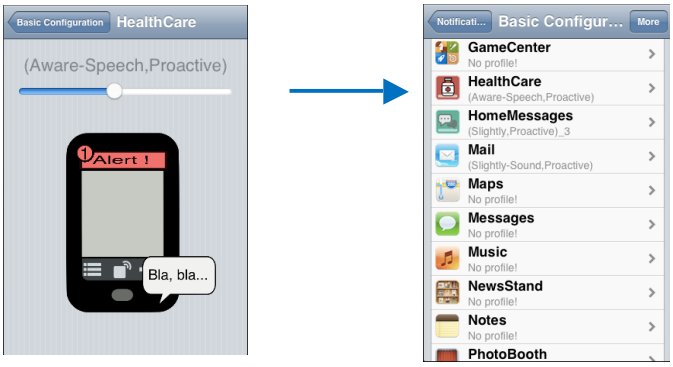
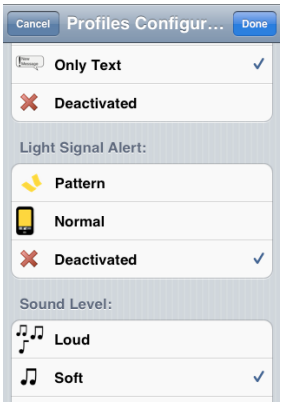


Figura 34. Creación de transición para servicio **HealthCare**



El otro perfil que intervendrá esta transición y se aplicará al servicio *HealthCare* cuando Bob **se encuentre comiendo en compañía**, será el perfil de nombre **(Slightly-Sound,Proactive)**. Dicho perfil alertará de manera más modesta mostrando un mensaje en pantalla *solo con* ***texto*** **acompañado de un sonido "*Beep*" a bajo volumen**. Se muestra la configuración de este perfil en la parte izquierda.

Creación de Transición:

Se crea una transición que reduzca el nivel de molestia cuando se dé la condición de que Bob no esté comiendo solo, es decir, cuando esté comiendo en compañía (*@LunchWithCompany*). Para ello se crea una transición siguiendo un modelo similar al del apartado anterior, que cambiará el perfil asociado por defecto al servicio *HealthCare* por el perfil *(Slightly-Sound,Proactive)* cuando se cumpla la condición descrita:

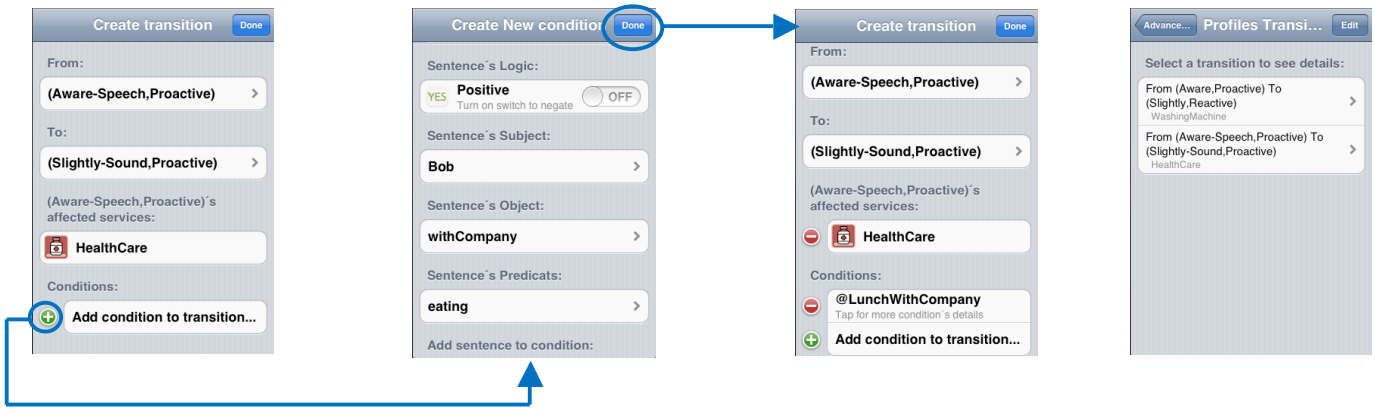
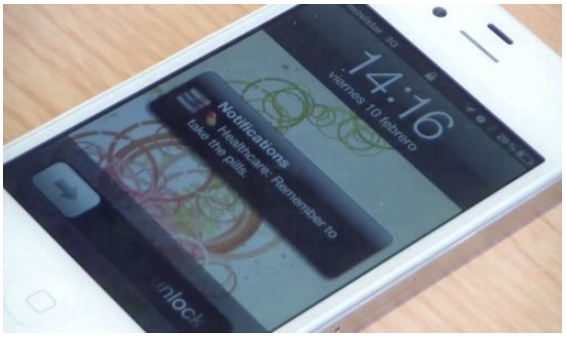


Figura 35. Asociación de perfil por defecto para servicio **Weather**

Mostrado de notificaciones:

A continuación se muestra como llegarían las notificaciones para este servicio. Por un lado se muestra el perfil por defecto que envía las notificaciones mostrando un mensaje en la parte superior con texto e icono y *Speech* (izquierda), mientras que por otra parte aparecen las notificaciones enviadas cuando Bob come en compañía, formadas por un mensaje acompañado de un sonido *Beep* de volumen bajo:



4.3 Weather

Este servicio monitoriza el tiempo meteorológico que hace en la zona donde se encuentra Bob, pudiendo avisar del tiempo que hace y aconsejar al usuario en función de éste a través de notificaciones.

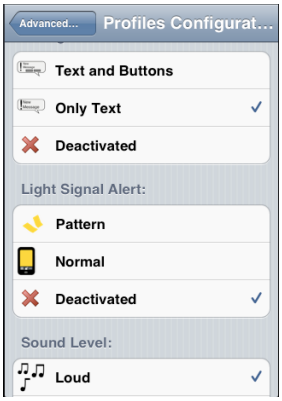
Descripción escenario:

Bob desea que por defecto las notificaciones se envíen **de manera invisible**, es decir, con un indicador (*badge*) en el icono de nuestra aplicación que no ofrece más información adicional acerca de la notificación entrante. Por otra parte desea **que si está saliendo de casa, solo y con el móvil en la mano**, las notificaciones se envíen de la manera más notoria, mostrando un mensaje sobre la notificación en pantalla acompañado de un sonido con el volumen alto.

Creación de Perfiles:

El perfil asociado por defecto al servicio Weather **(Invisible,Proactive)** tendrá todos los mecanismos de notificación desactivados, indicando así que sus notificaciones se representarán mediante un "badge" en el icono de nuestra aplicación.

Figura 36. Creación de transición para servicio **Weather**



El perfil al que se cambiará cuando se dé la condición comentada anteriormente se representará mediante el perfil **(Aware-Sound,Proactive)**, y avisará mostrando un texto en pantalla junto con un sonido con volumen alto. Podemos observar la configuración de dicho perfil en la imagen de la parte izquierda.

Creación de Transición:

La transición entre estos dos perfiles pretende que el usuario sea avisado con un nivel de alerta más alto cuando **éste sale de su casa, solo, y además lleva el móvil en la mano**. Para la creación de esta condición, que la llamaremos ***@leavingHomeAlone,*** hemos utilizado tres sentencias: ***Bob sale de su casa.*** ***Bob está solo.*** ***Lleva el móvil en la mano***. Haciendo así posible la adaptación del nivel de notificación a los requerimentos que el usuario Bob desee que se cumplan. Mostramos las capturas de la creación de esta transición ente estos perfiles y de la creación de la condición, con sus tres respectivas sentencias, que hará posible en intercambio de perfiles para el servicio Weather.

Una vez realizados estos pasos, la transición quedará creada:

Mostrado de notificaciones:

A continuación se muestra como llegarían las notificaciones según ambos perfiles. Por defecto se mostrarán como **un badge en el icono de la aplicación**, como se observa en la foto de la izquierda. Mientras que si se cumple la condición descrita, se enviarán utilizando un **mensaje de recomendación** (en este caso llevar paraguas ya que es muy probable que llueva) **en pantalla acompañado de un sonido con el volumen alto**, tal y como se muestra en la imagen de la parte derecha:

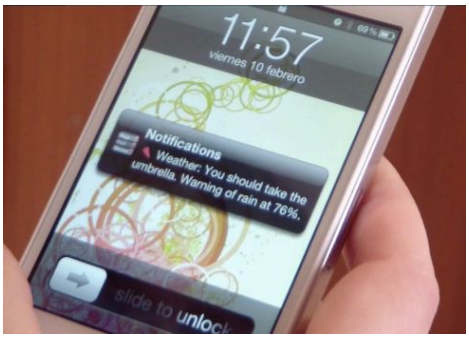


Figura 37. Asociación de perfil por defecto para servicio **Agenda**

4.4 Agenda

Este servicio actúa como un secretario personal del profesor Bob, permitiéndole crear citas y eventos y avisándole de todas sus citas diarias o próximas por medio de notificaciones.

Descripción escenario:

Bob desea que el sistema de notificación por defecto para este servicio sea un mensaje que se muestre en pantalla acompañado de un sonido *Beep.* Asimismo también quiere que por otra parte este aviso reduzca su nivel de alerta al **encontrarse en una reunión** y que aumente dicho nivel al **salir a correr con auriculares**. De esta manera se crearán **dos** transiciones: una para procesar el cambio de perfil cuándo se encuentra Bob en una reunión; y otra para variar el nivel de alerta cuando salga a correr escuchando música en sus auriculares.

Creación de Perfiles:

El perfil asociado a la agenda por defecto debe enviar sus notificaciones mostrando un mensaje en pantalla y emitiendo un sonido *Beep.* Podemos comprobar que ya disponemos de un perfil con este comportamiento: **(Slightly-Sound,Proactive)**, creado en el apartado correspondiente al servicio *HealthCare* que reutilizaremos para este servicio.

Para las transiciones de este servicio será necesario utilizar dos perfiles más, ya que habrá dos transiciones desde el perfil original. El primero de ellos debe enviar sus notificaciones de manera no intrusiva, por lo que Bob ha decidido utilizar el perfil creado para el servicio *Weather* **(Invisible,Proactive),** que enviará sus notificaciones de manera discreta colocando un indicador (badge) en el icono de la aplicación.

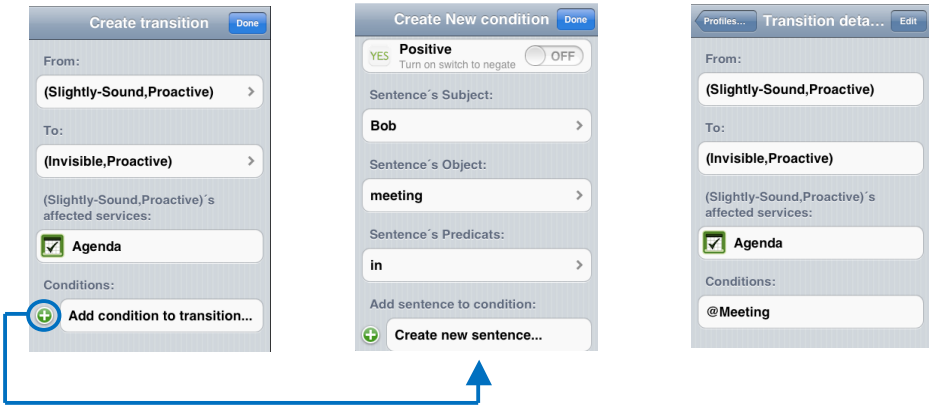
Por otra parte se ha utilizado otro perfil para que envíe las notificaciones de manera más sonora cuando el usuario salga a correr con sus auriculares, en este caso se mostrará un mensaje en la parte superior de la pantalla y el dispositivo "hablará" el contenido de la notificación haciendo uso de la funcionalidad *Speech.* Se observa que éste comportamiento coincide con el generado por el perfil **(Aware-Speech,Proactive)**, creado para el servicio *HealthCare,* por lo que se empleará este perfil para provocar la transición.

Figura 38. Creación de transiciones para servicio **Agenda**

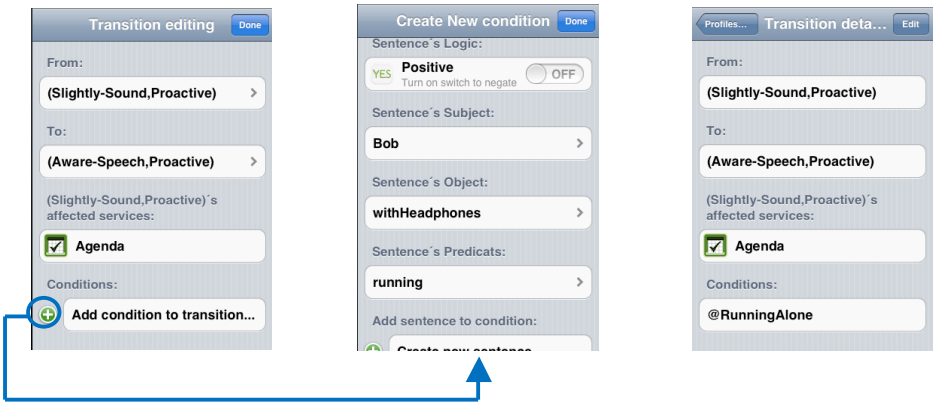
Creación de Transiciones:

Para suplir todas sus necesidades, Bob deberá crear **dos** transiciones a partir del perfil original. Pasaremos a describir estas transiciones a continuación:

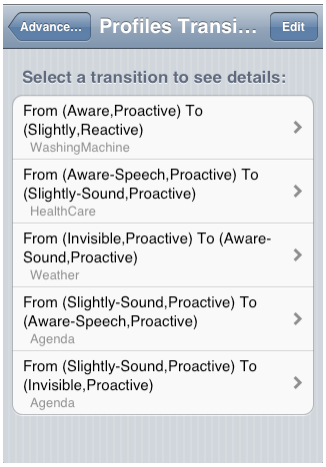
En primer lugar se debe crear una transición del perfil original al perfil **(Invisible,Proactive)** (que **silenciará** las notificaciones enviadas por el servicio *Agenda)* cuando se cumpla la condición de que **Bob esté en una reunión,** que hemos creado con el nombre *@Meeting***.** Se muestra imágenes de la creación de dicha transición:



En segundo lugar se crea la segunda transición para el caso en que **Bob salga a correr con sus auriculares puestos,** en cuyo caso se le asociará al servicio *Agenda* el perfil **(Aware-Speech,Proactive)** para que la notificación se envíe mediante un mecanismo de alerta más notable. Se muestra la creación de esta transición y su respectiva condición llamada *@RunningAlone* en la parte inferior:

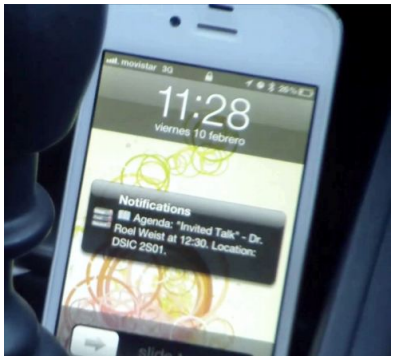


Así quedará todas las transiciones definidas en el sistema tras haber agregado las dos correspondientes al servicio Agenda:



Mostrado de notificaciones:

Por último indicaremos como se mostrarán en el terminal cada una de las diferentes notificaciones según el perfil de alerta que tenga asociado el servicio *Agenda* dependiendo del contexto actual. La imagen de la izquierda corresponde al perfil por defecto y envía las notificaciones con **un nivel de intrusión moderado,** un mensaje acompañado de un *Beep* que requiere esfuerzo por parte del usuario para ser leída; la imagen central se corresponde al **perfil no intrusivo** que se aplicará al servicio cuando Bob se encuentre en una reunión, tal y como se ha definido en la segunda transición; la imagen de la derecha corresponde al perfil con un mayor grado de intrusión, mostrando **un mensaje en la parte superior de la pantalla y una voz que lo lee (Speech)** y se utilizará cuando Bob salga a correr con sus auriculares:



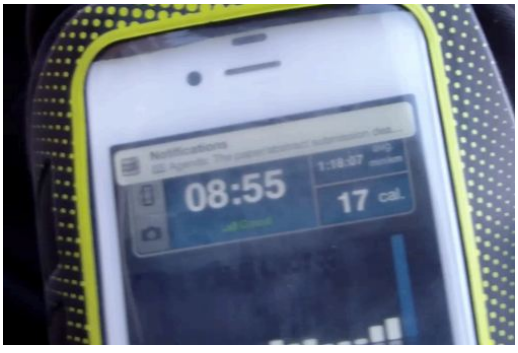


Figura 39. Asociación de perfil por defecto para servicio **Facebook**

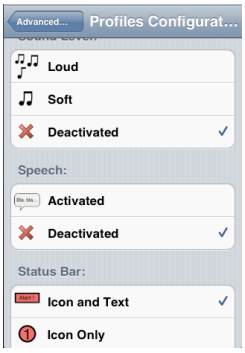
4.5 Facebook

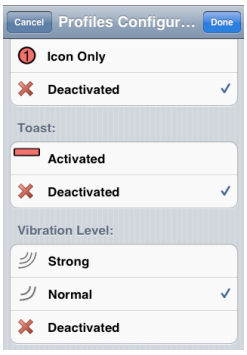
Este servicio avisa por medio de notificaciones cuando el usuario tiene una actividad determinada dentro de la red social *Facebook*, como puede ser: recibir un mensaje privado, que alguien escriba en su timeline, comentario en una foto, recibir un "me gusta", etc.

Descripción escenario:

Bob desea que por defecto estas notificaciones se envíen con una intromisión moderada, para que tenga que hacer esfuerzo para abrir la notificación y así se preserve la intimidad de los mensajes que puedan llegar. Por otra parte también tiene previsto que cuando esté dando un curso del que él es tutor, estas notificaciones se envíen de manera invisible, es decir, con un "badge" en el icono de la app.

Creación de Perfiles:





Se deberán utilizar dos perfiles para respetar estas normas. Un perfil por defecto que avise al usuario con un nivel de intromisión medio, para ello Bob crea un perfil **(Slightly-Vibrate,Proactive)** que **mostrará un mensaje de texto e icono en la parte superior de la pantalla que irá acompañado de una vibración con intensidad moderada**, tal y como se observa en las captura de la izquierda.

Este perfil deberá ser asignado al servicio *Facebook* como perfil por defecto:

Por otra parte se deberá usar otro perfil que avise al usuario de manera transparente, para ello utilizaremos el perfil **(Invisible,Proactive)**, ya existente, y que como hemos comentado anteriormente coloca un aviso (badge) en el icono de la app.

Figura 40. Creación de transición para servicio **Facebook**

Creación de Transiciones:

La condición que provocará que las notificaciones del servicio Facebook pasen a enviarse al usuario de manera discreta pasa por **comprobar si Bob se encuentra dando docencia de dicho curso,** por lo que la condición creada, *@Course,* deberá comprobar siesta circunstancia (sentencia) se cumple. Se muestran capturas de la transición entre los perfiles **(Slightly-Vibrate,Proactive)** y **(Invisible,Proactive)** que regirán el comportamiento de la aplicación *Facebook,* así como una imagen de como quedaría el sistema tras añadir esta transición.

Mostrado de notificaciones:

Se puede observar como llegarían estas notificaciones al terminal cumpliendo las restricciones impuestas por los perfiles que ha creado Bob. En la imagen de la izquierda llega una notificación por defecto, con un **mensaje en la parte superior con vibración**; mientras que cuando se lleve a cabo la transición, las notificaciones se enviarán **de manera transparente** tal y como se muestra en la captura de la derecha:

4.6 HomeMessages

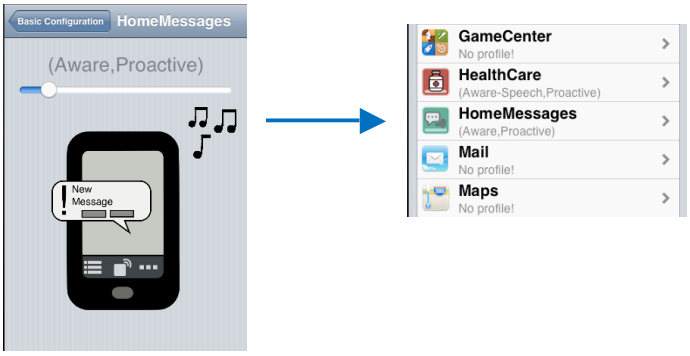
Este servicio avisa a Bob de mensajes que pueda enviar su familia desde su propia casa. Estas notificaciones se consideran importantes para Bob.

Descripción escenario:

Se pretende que por defecto estas notificaciones se envíen a Bob de manera sonora y visual, haciendo irremediable que éste preste atención al terminal. Dado su nivel de importancia, estas notificaciones solo reducirán su nivel de alerta cuando Bob se encuentre dando clase en el curso, por lo que habrá que crear una transición para tal condición.

Creación de Perfiles:

El perfil por defecto se corresponde con el comportamiento del perfil **(Aware,Proactive)** (que ya tendrá asociado otro servicio, *WashingMachine*), avisando mediante un **mensaje con botones en pantalla y aviso de sonido con volumen a nivel alto** que alertará a Bob de manera irremediable debido a la importancia del aviso. Se muestra como se asigna dicho perfil al servicio:



Por otra parte se deberá crear un perfil al que se cambiará cuando Bob esté dando clase, éste perfil alertará de manera indirecta mostrando un **mensaje de texto en pantalla junto con una vibración de nivel fuerte**, así Bob podrá darse cuenta que esa notificación proviene del servicio *HomeMessages* sin necesidad de fijar su atención en el dispositivo. A continuación se muestran imágenes de la creación del perfil **(Aware-Vibrate,Proactive)** que cumple con los requisitos expuestos y usaremos como perfil destino en esta transición:

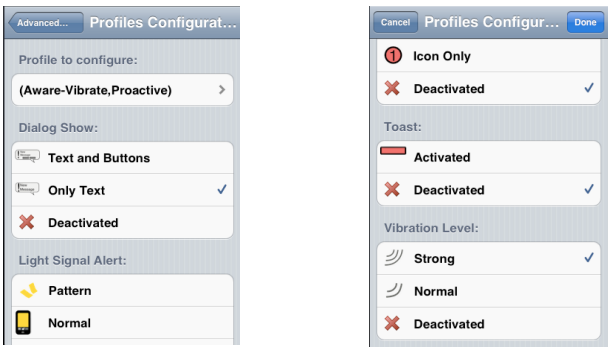
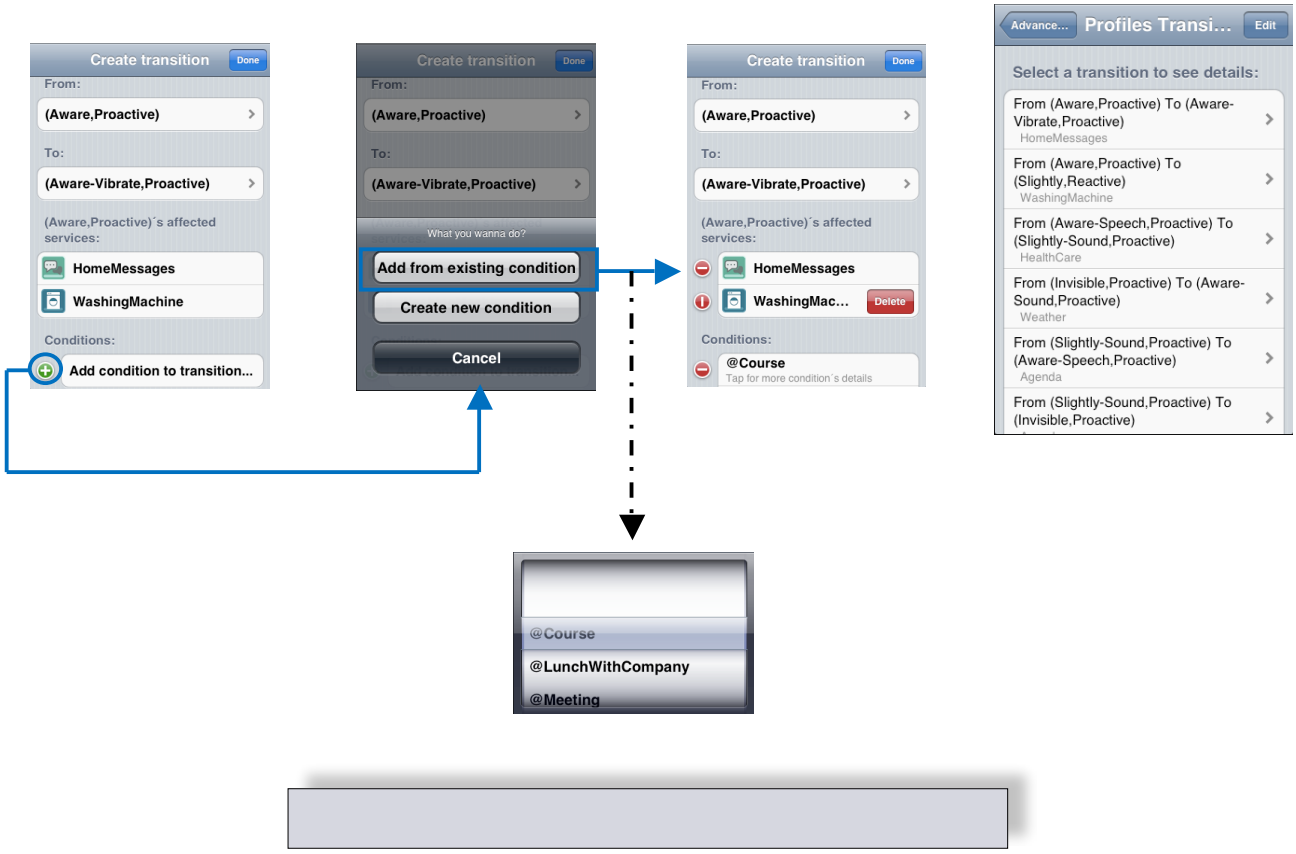


Figura 42. Creación de transición para servicio **HomeMessages**

Creación de Transiciones:

Se deberá crear una transición desde el perfil de nombre **(Aware,Proactive)** hacia el perfil **(Aware-Vibrate,Proactive)** para disminuir así el nivel de atención mientras Bob se encuentre impartiendo docencia en el curso según lo expuesto en el apartado anterior. Para ello utilizaremos la condición ya existente *@Course,* que desencadenará el cambio de perfil. A continuación se ven capturas de la creación de dicha transición, se puede observar que el perfil **From** ya posee de otro servicio asociado, *WashingMachine*, que deberemos eliminar de esta transición:



Mostrado de notificaciones:

A continuación se muestra la manera en la que se le molesta al usuario respecto a éste último perfil de molestia **(Aware-Vibrate,Proactive)**, que como vemos mostrará un **mensaje de texto junto con vibración a nivel máximo**:

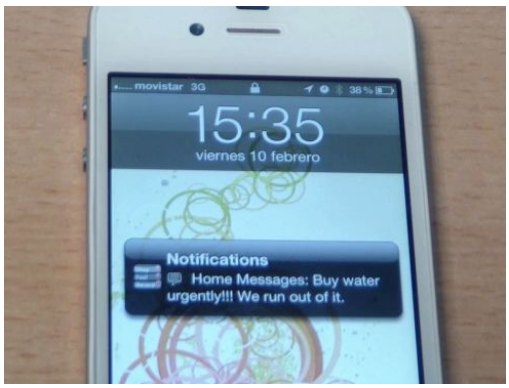


Figura 43. Asociación de perfil por defecto para servicio **Shopping**

4.7 Shopping

Esta aplicación permite crear una lista de productos de la compra con una prioridad determinada. También sugiere productos que podrían interesar al usuario en función de sus intereses. Además de enviar notificaciones cuando sea urgente comprar algún producto o se presente un producto recomendado.

Descripción escenario:

Bob pretende que ésta aplicación envíe las notificaciones de manera invisible de manera predeterminada, consiguiendo así que mientras esté trabajando o durmiendo no sea molestado. Por otro parte, cuando pase cerca de un supermercado quiere que la notificación sea **visual y sonora (mediante Speech)** para percatarse si se le ha olvidado comprar algún producto necesario para su día a día.

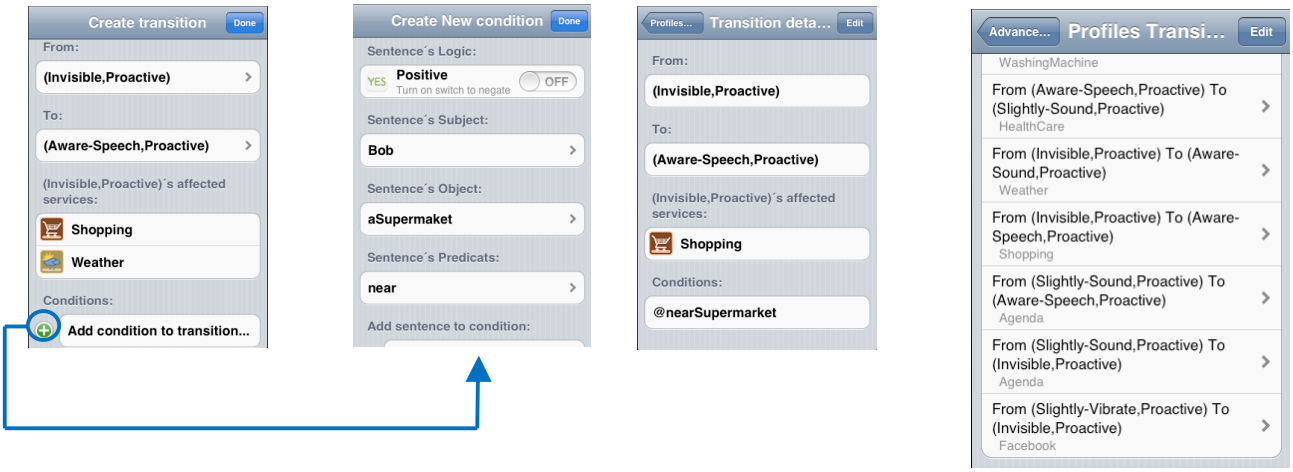
Creación de Perfiles:

Para los requerimentos de notificación de este servicio, Bob decide utilizar dos perfiles creados con anterioridad. Por una parte el perfil **(Invisible,Proactive),** que representará el perfil con alerta de notificaciones invisible y será asociado por defecto al servicio. Y por otro lado el perfil **(Aware-Speech,Proactive),** que notificará al usuario mediante un mensaje en la parte superior de la pantalla acompañado de un sonido *Speech:*

Creación de Transiciones:

Como hemos comentado, Bob quiere que el teléfono avise de manera discreta por defecto salvo cuando éste pase cerca de un supermercado solo en el coche, en cuyo caso, si llega una notificación de este servicio, aparecerá de manera más intrusiva en el teléfono mostrando un mensaje con sonido. Se deberá crear una transición para esta condición, que hemos llamado *@nearSupermarket,* y provocará un cambio de perfil según este comportamiento:

Figura 44. Creación de transición para servicio **Shopping**



Mostrado de notificaciones:

Por último mostraremos la manera en que se presentan estas notificaciones al usuario en cada uno de los perfiles asociados a este servicio. La imagen de la izquierda corresponde al comportamiento por defecto, en que los avisos llegan **de manera invisible en forma de *Badge*** en el icono de la app, para no molestar mientras esté durmiendo por ejemplo. La captura de la derecha presenta la manera en que se notificará a Bob cuando éste pase cerca de un supermercado y haya algún producto que sea urgente comprar, en cuyo caso se mostrará un **mensaje en la parte superior de la pantalla** cuyo contenido será "hablado" mediante ***Speech**:*

5 Conclusiones y ampliaciones

Ya hemos expuesto todas las tareas que realiza nuestra aplicación, y la manera en como se llevan a cabo. En este último capítulo de la memoria, explicaremos las conclusiones obtenidas de dicho proyecto, también expondremos una serie de ampliaciones que se puedan implementar en la aplicación en un futuro para mejorar su funcionalidad o interfaz gráfica.

5.1 Conclusiones sobre el proyecto

Con la realización de este proyecto he adquirido varios conocimientos que expondré a continuación: En primer lugar he aprendido la sintaxis y utilización del lenguaje de programación Objective-C, abriendo nuevos campos y posibilidades en el desarrollo de aplicaciones. Siendo esto bastante útil en la vida laboral ya que hay un mercado en continua expansión de programación de aplicaciones para smartphones y tabletas.

Este conocimiento abre un amplio abanico de posibilidades profesionales relacionadas con la programación móvil. Centrándose en el mundo del desarrollo del sistema iOS, en el que he adquirido bastante base y experiencia para poder empezar a formarme de manera avanzada en el ecosistema de creación de aplicaciones, sentando así las bases para poder orientar este conocimiento de manera profesional.

Por otra parte la aplicación creada resulta bastante útil en este contexto de mercado de aplicaciones móviles. Ya que cada vez van apareciendo más y más aplicaciones que envían notificaciones al usuario; así como nuevos sistemas y mecanismos en los sistemas operativos móviles para hacer llegar estos avisos al usuario. Por lo que resulta muy atractiva la idea de disponer de una aplicación que haga de centro de notificaciones, permitiendo configurar de manera avanzada todas y cada una de las notificaciones que envían las diferentes aplicaciones instaladas en el teléfono. Siendo esta funcionalidad de control de notificaciones un tipo (género) de aplicaciones que aun no está muy explotado, y las aplicaciones existentes en el mercado tienen una interfaz confusa que en muchas ocasiones conduce al error.

También, al coger soltura con el lenguaje Objective-C, podría empezar a interesarme y formarme en la programación de aplicaciones de sobremesa para el sistema operativo Mac OSX. Que además de compartir entorno de desarrollo (Xcode) comparte algunas librearías y funciones básicas con iOS. Lo que hacen relativamente sencillo el salto de desarrollo de un sistema a otro.

5.2 Posibles ampliaciones de la aplicación

A continuación expondremos varias maneras en las que podríamos ampliar la utilidad de nuestra aplicación:

- **Utilizar Geolocalización:** Podríamos ampliar la funcionalidad de nuestra app añadiendo la posibilidad de crear sentencias para las condiciones en función de la localización, utilizando para ello las librerías que nos ofrece iOS para el acceso al sistema de GPS.

- **Implementación de perfiles de usuario:** En vez de tener una configuración global de perfiles, condiciones y transiciones, se podría implementar un sistema de gestión multiusuario. Este sistema debería guardar una configuración personalizada para cada uno de los usuarios. Creando también los mecanismos necesarios para el inicio y cierre de sesión de usuario en la aplicación. También se debería de almacenar la información por cada usuario respectivamente en las tablas de la base de datos del servidor.

- **Envío de notificaciones de nuestra aplicación:** Podríamos hacer que nuestra aplicación enviara notificaciones auxiliares propias, a través de centro de notificaciones nativo de iOS, cada vez que produjera una transición, o se estableciese un perfil al ejecutar una aplicación. Sería una mejora que haría más agradable y clara la interacción del usuario con la interfaz gráfica de nuestra aplicación.

- **Sincronización con iCloud:** Podríamos utilizar el framework que nos ofrece iOS para, por ejemplo, importar los contactos (o parte de ellos) como Objetos y Sujetos para las sentencias de la aplicación. O podríamos sincronizar parte de la configuración interna de nuestra aplicación con iCloud, para aliviar así la carga del servidor dedicado que ofrecemos.

- **Mejoras visuales en la interfaz:** Podríamos mejorar nuestra aplicación también ofreciendo una interfaz gráfica más bonita y agradable que haga más amena la navegación por los diferentes menús. Esto es una tarea orientada al diseño en la que sería aconsejable trabajar junto con la ayuda un diseñador gráfico.

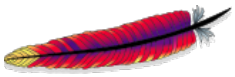
6 Anexo - Software utilizado en parte servidor y software adicional empleado

Nuestra aplicación tiene dos partes bien diferenciadas: la parte del cliente, que representa la propia aplicación corriendo desde el iPhone, y la parte servidor, que se encarga del almacenamiento y persistencia con el tiempo de los datos.

Esta arquitectura cliente-servidor requiere de la instalación y correcta configuración de multitud de software orientado la parte del servidor: como el propio servidor web, Apache, el servidor de base de datos SQL, el demonio MySQL, y otros elementos igualmente importantes como es la habilitación de la tecnología PHP en el servidor Apache, JSON como medio para la codificación de datos en el servidor, o el software adicional utilizado para comprobar el correcto funcionamiento del servidor creado.

En este anexo analizaremos toda la instalación y configuración necesarias de estas herramientas para llegar al punto de disponer de un servidor usable y estable, capaz de comunicarse con la aplicación creada:

6.1 Servidor Web Apache



Apache ha sido uno de los servidores más populares, configurable, moldeable, estable y utilizado por multitud de empresas y particulares durante todos los tiempos hasta el día de hoy. Esto se debe a que es un proyecto libre, de código abierto, que ha contado con el apoyo de la comunidad durante mucho tiempo. Este trabajo y apoyo ha dado lugar a una herramienta servidor muy potente y versátil que analizaremos durante este primer punto del anexo.

6.1.1 Descripción y características

Apache es el servidor web más utilizado desde 1996, fue un proyecto iniciado por la Apache Software Foundation con el nombre de httpd [26]. Apache, para la publicación tanto de su software como de su documentación, utiliza varias licencias. La versión de esta licencia es la "*Apache License, 2.0*" que es compatible con la licencia GPL [27], es decir, permite ser utilizado para crear software con cualquier propósito, modificarlo, y distribuirlo. Pero el software creado a partir de uno con licencia GPL no tiene porqué ser distribuido con la misma licencia.

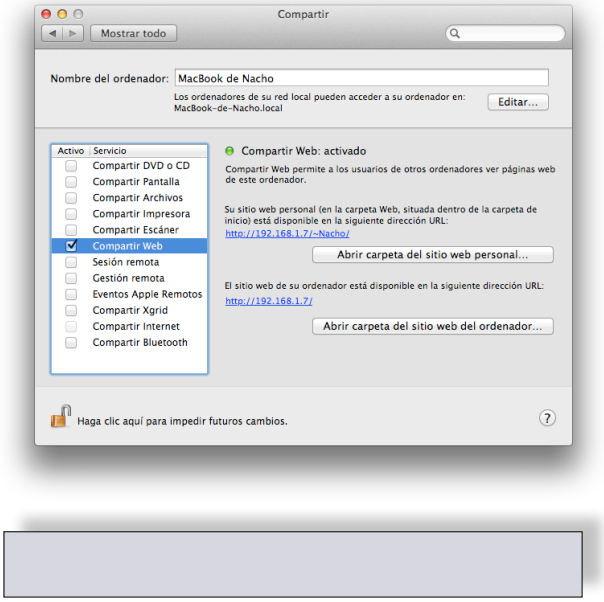
Este software puede funcionar en infinidad de sistemas operativos, tanto Windows, Mac OSX, Linux y cualquier variante UNIX que se pueda encontrar.

Tiene una **arquitectura modular**, en la que el núcleo guarda la funcionalidad básica del servidor, y luego el usuario puede añadir o quitar funcionalidades a su antojo utilizando módulos para ello. Los módulos son como "plugins" que extienden la funcionalidad inicial. Esto provoca que Apache sea ampliamente flexible y configurable a las diferentes necesidades que tenga cada usuario o empresa.

6.1.2 Instalación y configuración

En este punto veremos la manera de instalar Apache en un equipo con el sistema operativo Mac OSX y en otro con el sistema Linux instalado, en concreto la versión 11.10 de Kubuntu [28].

Mac OSX:



Apache es, como hemos dicho antes, el más usado de los servidores web. Prueba de ello es que viene preinstalado de serie en Mac OSX, teniendo el usuario solamente que activar el servicio **compartir web** en el apartado *compartir* de *panel de preferencias del sistema*, tal y como se puede observar en la figura 14 en la izquierda, para comenzara utilizarlo. La configuración se realiza utilizando un fichero de texto con extensión .conf. Este fichero tiene el nombre de **httpd.conf** y su se encuentra en la ruta: **/private/etc/apache2.** En este fichero le indicamos al servidor los módulos que queremos que cargue al inicio y los que no , la dirección IP y puerto TCP donde queremos que escuche, como queremos que abra ciertos tipos de ficheros, privacidad de directorios… Todas estas funciones se pueden además extender agregando nuevos módulos al servidor e incluyéndolos en este fichero de texto. A continuación se muestra una captura de dicho fichero que muestra la parte de módulos activados y desactivados (por ejemplo está desactivado el que empieza por el carácter #, en línea 95); nótese también que hemos activado el módulo **PHP**, que por defecto viene desactivado, ya que lo usaremos en nuestro servidor como "puente" entre la aplicación y la base de datos, hablaremos de él más adelante:

Figura 46. Fichero de configuración **/private/etc/apache2/httpd.conf**

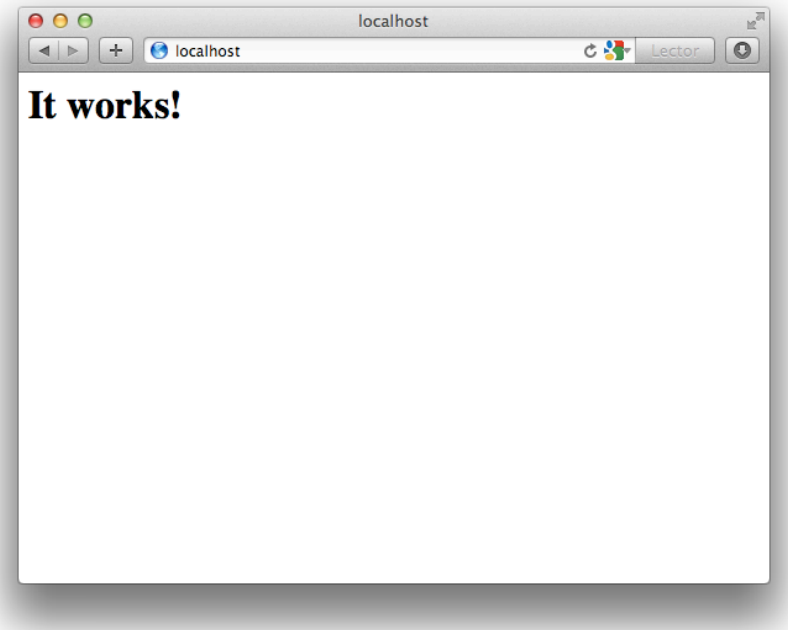
```
93  LoadModule proxy_scgi_module libexec/apache2/mod_proxy_scgi.so
94  LoadModule proxy_balancer_module libexec/apache2/mod_proxy_balancer.so
95  #LoadModule ssl_module libexec/apache2/mod_ssl.so
96  LoadModule mime_module libexec/apache2/mod_mime.so
97  LoadModule dav_module libexec/apache2/mod_dav.so
98  LoadModule autoindex_module libexec/apache2/mod_autoindex.so
99  LoadModule asis_module libexec/apache2/mod_asis.so
100 LoadModule info_module libexec/apache2/mod_info.so
101 LoadModule cgi_module libexec/apache2/mod_cgi.so
102 LoadModule dav_fs_module libexec/apache2/mod_dav_fs.so
103 LoadModule vhost_alias_module libexec/apache2/mod_vhost_alias.so
104 LoadModule negotiation_module libexec/apache2/mod_negotiation.so
105 LoadModule dir_module libexec/apache2/mod_dir.so
106 LoadModule imagemap_module libexec/apache2/mod_imagemap.so
107 LoadModule actions_module libexec/apache2/mod_actions.so
108 LoadModule speling_module libexec/apache2/mod_speling.so
109 LoadModule alias_module libexec/apache2/mod_alias.so
110 LoadModule rewrite_module libexec/apache2/mod_rewrite.so
111 LoadModule php5_module libexec/apache2/libphp5.so
```

También es conveniente indicar que para detener, o volver a iniciar el servidor Apache, aparte de obviamente poder hacerlo activando y desactivando el servicio *compartir web* en el *panel de preferencias del sistema*, también lo podemos hacer introduciendo alguno de los siguientes comandos en el terminal, sirviendo cada uno para reiniciar, iniciar o parar el servidor respectivamente:

Figura 16. Finalización exitosa de instalación de Apache

```
> sudo apachectl -k restart

> sudo apachectl -k start

> sudo apachectl -k stop
```

Una vez tenemos todo configurado a nuestro gusto, para comprobar que todo funciona como es debido, podemos introducir la dirección de nuestra máquina en un navegador web. Nos aparecerá la siguiente imagen si al menos la configuración básica se ha realizado correctamente:



LINUX:

La instalación el servidor web Apache en una máquina Linux varía con respecto a la de Mac OSX en dos principales aspectos: tenemos que tener la configuración un poco más en cuenta, además de tener que descargar el programa e instalarlo de uno de los repositorios disponibles (*aptget* por ejemplo). También existe la posibilidad de descargar el código fuente de la web de Apache y hacer una instalación más personalizada compilando el código fuente por cuenta del usuario. Para descargarlo debemos introducir el siguiente comando en una consola con permisos de superusuario:

```
# sudo apt-get install apache2
```

Una vez instalado podremos configurarlo de la misma manera que Mac OSX a través del ficher **httpd.conf**. En cualquier distribución arrancaremos el servidor con los mismos comandos de terminal expuestos para Mac OSX.

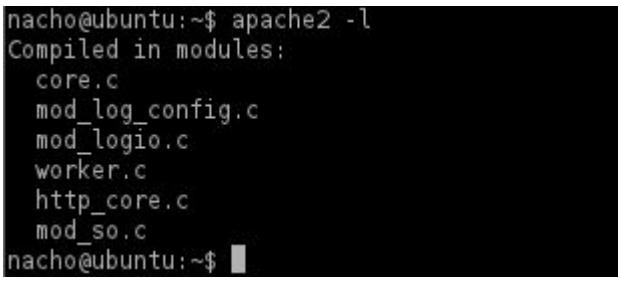
Podemos ver los ficheros de **log** con los siguientes comandos de terminal, que muestran lo último ocurrido en dichos ficheros (tail: orden para leer las últimas líneas de un fichero):

```
# tail –f /var/www/apache2/access.log
# tail –f /var/www/apache2/error.log
```

También deberemos activar el módulo **PHP** descomentando la respectiva línea y añadir las siguientes líneas de código al fichero *httpd.conf* para activar la extensión .php y permitir la interpretación de ficheros con esta extensión:

```
<Files *.php>
        SetOutputFilter PHP
        SetInputFilter PHP
</Filter>
```

Tanto en Mac OSX como en LINUX podemos comprobar los módulos que vienen cargados por defecto en nuestra instalación de Apache con el siguiente comando de terminal:

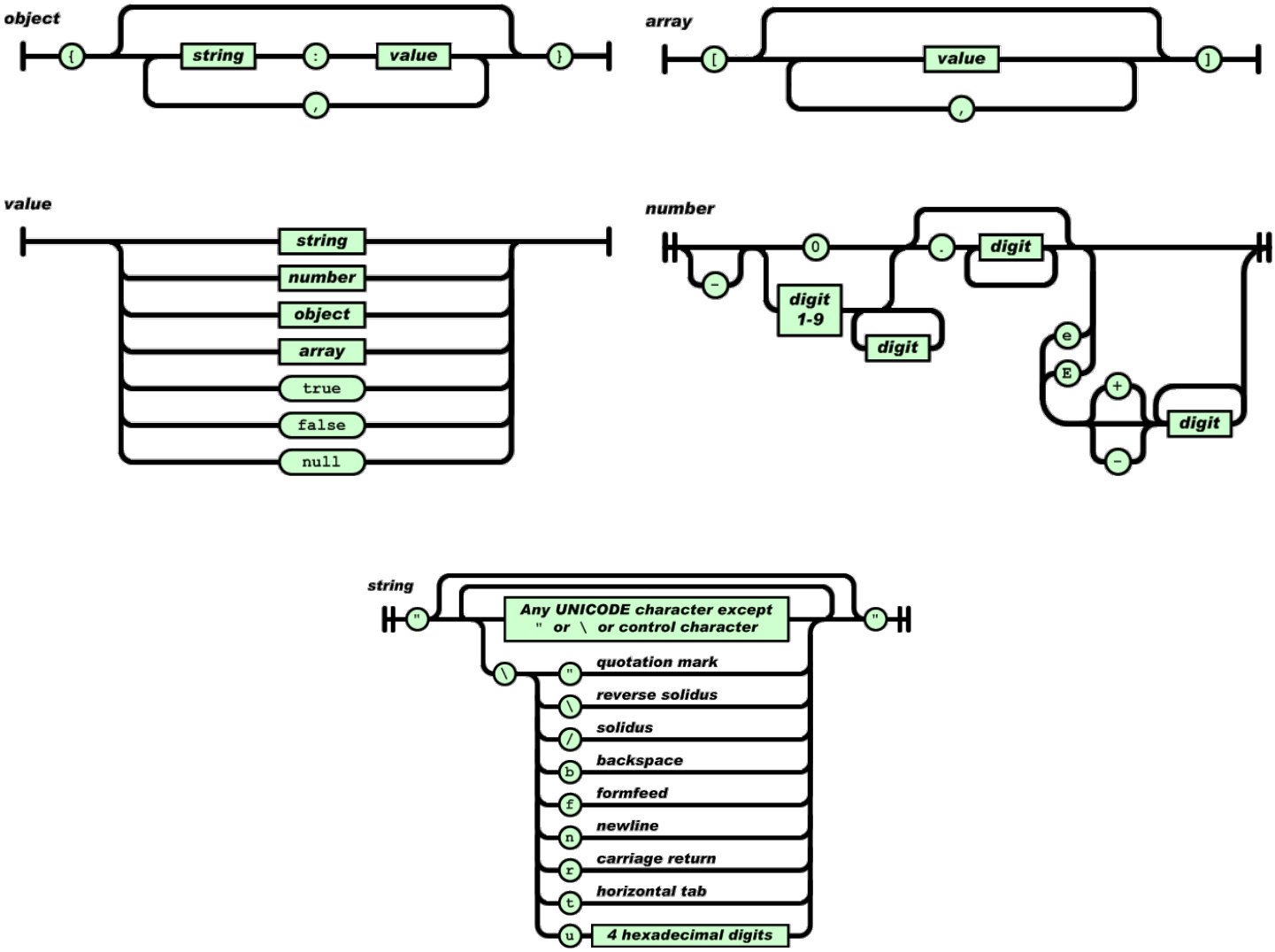


6.2 Codificación de objetos JSON

Figura 47. Objetos representables mediante JSON

Antes de continuar explicando las herramientas del servidor, es necesario incidir en el método de codificación de objetos **JSON** (Java Script Object Notation) [29]. JSON un formato ligero de codificación para hacer posible el intercambio de datos entre el cliente y el servidor. JSON se ofrece como alternativa a XML en muchos escenarios, ya que es una notación de objetos JavaScript [30] independiente a este lenguaje. La principal ventaja de JSON frente a XML es su sencillez, que lo dotan a su vez de gran potencial, permitiendo que escribir un analizador sintáctico (*parser*) de JSON sea una tarea relativamente sencilla. JSON se emplea habitualmente en entornos donde el tamaño del flujo de datos entre el cliente y servidor es de vital importancia, por ejemplo Google, Yahoo, Twitter o Facebook lo utilizan

Se pueden representar gran cantidad de estructuras de datos mediante este sistema, como por ejemplo: Valores simples, Strings, Arrays, numbers, Diccionarios con sus respectivos pares clave valor, o incluso el número *e*. Estos objetos se representan según el formato expuesto en la imagen a continuación:



Mostraremos en la parte inferior un ejemplo práctico de esta codificación: consiste en una respuesta en formato JSON obtenida de la red social Twitter, facilitando así el acceso al desarrollador al contenido de dicha red social. En esta respuesta se agrupan unos valores según sus respectivas claves, éste suele ser el escenario de uso clásico de JSON:

[{"created_at":"Tue Jul 10 01:42:10 +0000 2012","id":222505965281488898,"id_str":"222505965281488898","text":"We will be performing #D3 maintenance Tuesday, July 10 beginning at 5am PDT: http:\/\/t.co\/2jr91xDu","source":"\u003ca href=\"http:\/\/www.radian6.com\" rel=\"nofollow\"\u003eRadian6 \u003c\/a\u003e","truncated":false,"in_reply_to_status_id":null,"in_reply_to_status_id_str":null,"in_reply_to_user_id":null,"in_reply_to_user_id_str":null,"in_reply_to_screen_name":null,"user":{"id":174307074,"id_str":"174307074","name":"BlizzardCS","screen_name":"BlizzardCS","location":"Irvine CA and Austin TX","description":"Blizzard Entertainment North America CustomerSupport","url":"http:\/\/blizzard.com\/support\/","protected":false,"followers_count":220559,"friends_count":32,"listed_count":2008,"created_at":"Tue Aug 03 16:26:52 +0000 2010}]

Podemos observar que esta respuesta de la API de Twitter devuelve un vector (array) formado por un diccionario de objetos (tuplas clave valor) cuyo valor podremos consultar desde cualquier aplicación capaz de interpretar datos en formato JSON.

6.3 PHP

6.3.1 Descripción y características

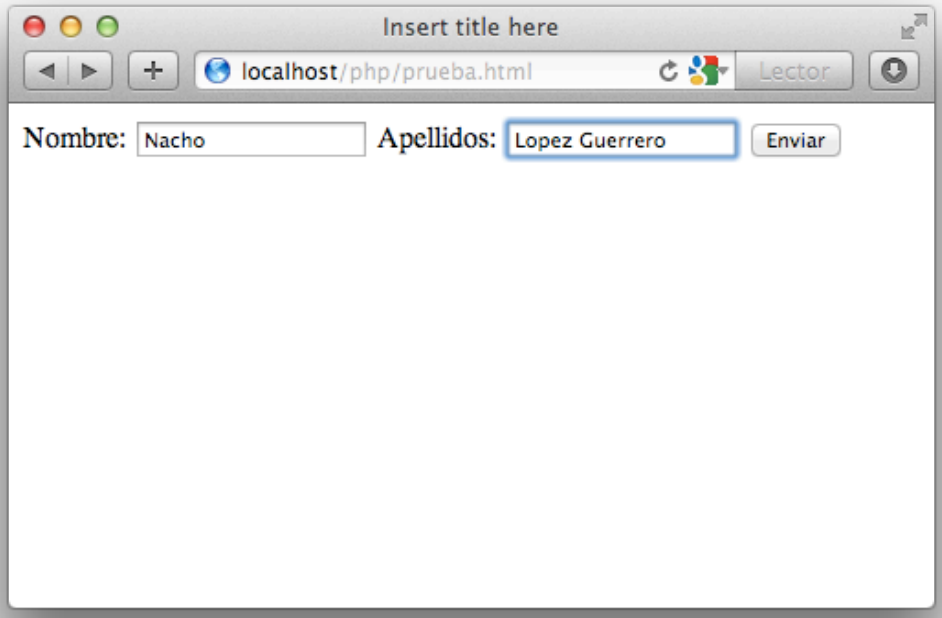
En el punto actual en el que nos encontramos nuestro servidor Apache es capaz de mostrar perfectamente contenido a los usuarios que se conecten a nuestra máquina, pero este contenido será exclusivamente estático, por lo que tendríamos que cambiar "a mano" el código fuente de la web cada vez que deseáramos añadir algo. Además también precisaremos de un mecanismo para capturar los datos que necesite en cliente.

PHP es un lenguaje interpretado (no precisa de compilación) con una sintaxis muy familiar y sencilla para el programador que se utiliza para la ejecución de scripts en la parte del servidor. El código PHP se integra con el código HTML de una página Web, permitiendo así crear partes dinámicas de contenido en nuestra página. Este lenguaje ofrece también soporte nativo para interactuar con la mayoría de servidores de bases de datos MySQL, Postgres, Oracle o Microsoft SQL Server, pudiendo así comunicarse con un SGDB y hacer consultas (querys) sobre él. También hace posible el envío de datos por parte del cliente a nuestro servidor por medio de los métodos **GET** y **POST** (guardando estos datos en las variables **$_GET** y **$_POST** respectivamente).

El código PHP nunca debe de ser visible para el usuario (cliente) que se conecta a nuestro servidor, ya que este código se ejecuta dentro de nuestra máquina servidor y podría dar lugar a una brecha de seguridad. Para ello PHP se incrusta dentro del código HTML y al ejecutar un script PHP nuestro servidor devuelve un fichero HTML conteniendo la respuesta resultante de ejecutar el script PHP incrustado en dicho fichero. Gracias a este mecanismo, el cliente recibirá únicamente la salida de dicho script incluida en un fichero HTML regular.

A continuación mostramos un ejemplo del funcionamiento de este mecanismo: constará de un fragmento de código HTML (fichero prueba.html) y un script PHP (prueba.php). El primero es una página web que pregunta al usuario por su nombre y apellidos y, una vez que éste los introduce y pulsa el botón "Enviar", son enviados al servidor dentro de la variable **$_GET,** donde se ejecutará el script *prueba.php*, del que hablaremos también a continuación:

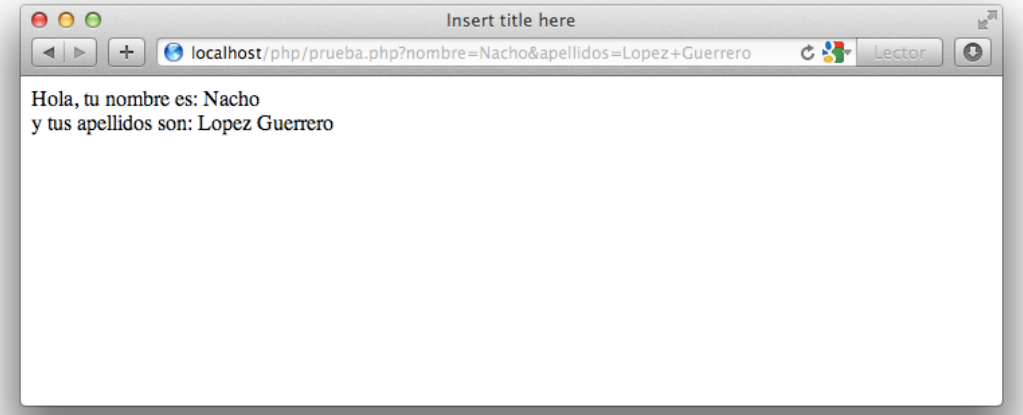
```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN" "http://www.w3.org/TR/
   html4/frameset.dtd">
2  <html>
3  <head>
4  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
5  <title>Insert title here</title>
6  </head>
7  <body>
8  <form action="prueba.php" method="GET">
9  Nombre: <input type="text" name="nombre"></input>
10 Apellidos: <input type="text" name="apellidos"></input>
11 <input type="submit">
12 </form>
13 </body>
14 </html>
```



Una vez el usuario aprieta el botón “Enviar”, se ejecuta el siguiente script en la parte de servidor utilizando como argumentos los datos haya introducido el cliente. Estos datos se enviarán como hemos dicho en la variable $_GET. El script utiliza la orden *echo* para imprimir un mensaje por la pantalla en función de estos argumentos recibidos. Se puede observar que el código PHP se encuentra entre las llaves **<?php** **?>**

```
1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN">
2  <html>
3  <head>
4  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
5  <title>Insert title here</title>
6  </head>
7  <?php
8      echo "Hola, tu nombre es: " . $_GET['nombre'] . "<br>";
9      echo "y tus apellidos son: " . $_GET['apellidos'] . "<br>";
10 ?>
11
12 </html>
```

El fichero prueba.php se ejecutará y creará un fichero HTML que recibirá el cliente con el siguiente contenido generado en consecuencia. Este contenido habrá sido creado en función de los datos que se hayan introducido, que como se puede observar en la barra de direcciones, se envían junto con la dirección del script:



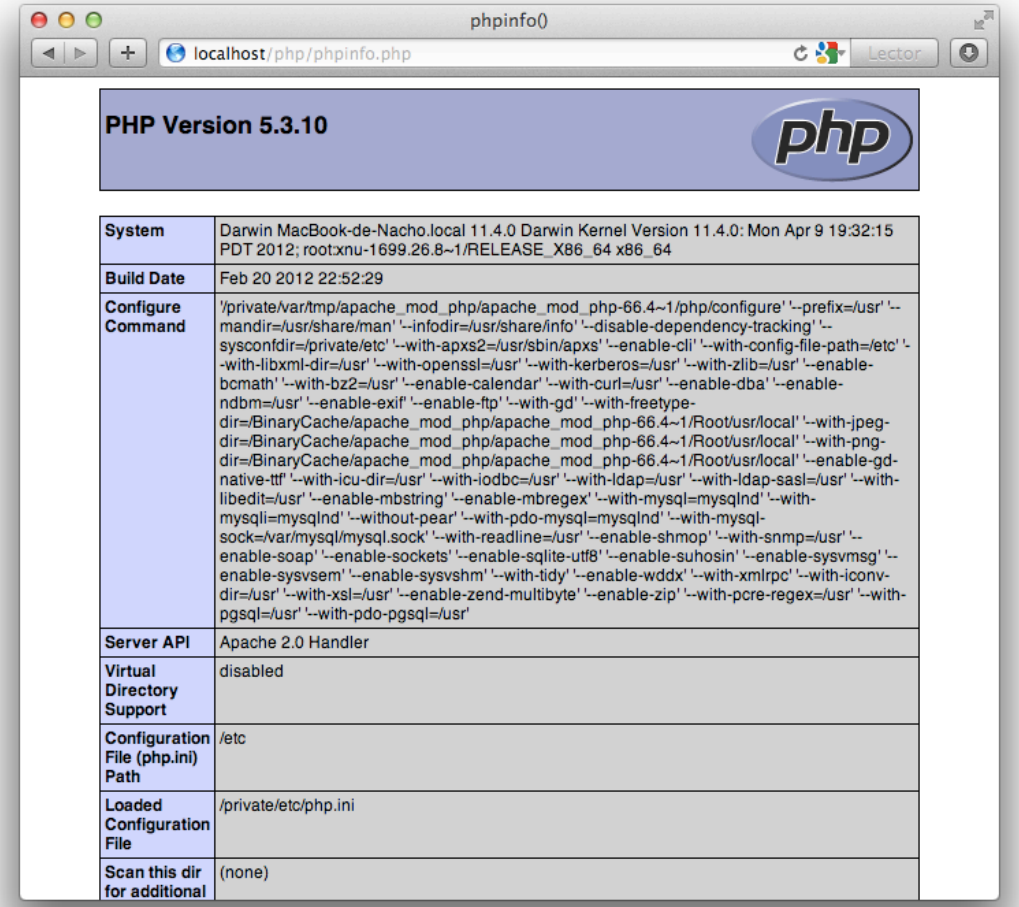
6.3.2 Instalación y configuración

La instalación de PHP es común a Mac OSX y LINUX, consiste en incluir en nuestro servidor Apache un módulo, indicándolo así en nuestro fichero de configuración (*httpd.conf*), tal y como hemos mencionado en el punto 2.2.1. En concreto el módulo que debemos utilizar es el siguiente (nombre y ruta de fichero de módulo) nótese que utilizamos la versión 5.3.10 de PHP en nuestro servidor:

```
php5_module libexec/apache2/libphp5.so
```

Para realizar la configuración de PHP, debemos modificar un fichero llamado **php.ini** que se encuentra en el directorio **/etc.** En este fichero indicaremos cómo se debe comportar php utilizando una serie de directivas para ello (parecido al caso de Apache). Dentro de este fichero podemos indicar información tan variada como: la zona horaria, la posibilidad de elegir si mostramos los errores o no, activación de ficheros de logs, numero máximo de errores soportados, el tamaño máximo de los datos enviados mediante el método POST o el tamaño máximo de fichero que se pueden subir al servidor.

Una vez tenemos todo configurado, podemos crear un script que llame a la función **phpinfo()** y si todo está debidamente configurado, nos debería de mostrar la información de nuestra distribución de PHP instalada, tal y como muestra la siguiente captura:



System	Darwin MacBook-de-Nacho.local 11.4.0 Darwin Kernel Version 11.4.0: Mon Apr 9 19:32:15 PDT 2012; root:xnu-1699.26.8~1/RELEASE_X86_64 x86_64
Build Date	Feb 20 2012 22:52:29
Configure Command	'/private/var/tmp/apache_mod_php/apache_mod_php-66.4~1/php/configure' '--prefix=/usr' '--mandir=/usr/share/man' '--infodir=/usr/share/info' '--disable-dependency-tracking' '--sysconfdir=/private/etc' '--with-apxs2=/usr/sbin/apxs' '--enable-cli' '--with-config-file-path=/etc' '--with-libxml-dir=/usr' '--with-openssl=/usr' '--with-kerberos=/usr' '--with-zlib=/usr' '--enable-bcmath' '--with-bz2=/usr' '--enable-calendar' '--with-curl=/usr' '--enable-dba' '--enable-ndbm=/usr' '--enable-exif' '--enable-ftp' '--with-gd' '--with-freetype-dir=/BinaryCache/apache_mod_php/apache_mod_php-66.4~1/Root/usr/local' '--with-jpeg-dir=/BinaryCache/apache_mod_php/apache_mod_php-66.4~1/Root/usr/local' '--with-png-dir=/BinaryCache/apache_mod_php/apache_mod_php-66.4~1/Root/usr/local' '--enable-gd-native-ttf' '--with-icu-dir=/usr' '--with-iodbc=/usr' '--with-ldap=/usr' '--with-ldap-sasl=/usr' '--with-libedit=/usr' '--enable-mbstring' '--enable-mbregex' '--with-mysql=mysqlnd' '--with-mysqli=mysqlnd' '--without-pear' '--with-pdo-mysql=mysqlnd' '--with-mysql-sock=/var/mysql/mysql.sock' '--with-readline=/usr' '--enable-shmop' '--with-snmp=/usr' '--enable-soap' '--enable-sockets' '--enable-sqlite-utf8' '--enable-suhosin' '--enable-sysvmsg' '--enable-sysvsem' '--enable-sysvshm' '--with-tidy' '--enable-wddx' '--with-xmlrpc' '--with-iconv-dir=/usr' '--with-xsl=/usr' '--enable-zend-multibyte' '--enable-zip' '--with-pcre-regex=/usr' '--with-pgsql=/usr' '--with-pdo-pgsql=/usr'
Server API	Apache 2.0 Handler
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc
Loaded Configuration File	/private/etc/php.ini
Scan this dir for additional	(none)

En nuestra aplicación utilizamos scripts **PHP** para enviar/recibir los datos que se usan la interfaz gráfica. Estos datos viajan codificados en JSON (gracias a la función *json_encode()*) para que sean interpretados por la parte del cliente de manera correcta. De ahí la necesidad de explicar **JSON** en el punto anterior antes de PHP.

Figura 51. Módulo **MySQL Community Server**

6.4 Servidor de base de datos MySQL

En este punto 6.4 del anexo explicaremos las principales características, instalación y configuración necesaria del sistema de gestión de base de datos (SGBD) elegido para la realización de este proyecto: **MySQL**. Hemos elegido este sistema ya que es el que más familiar resulta, y es compatible con la gran mayoría de SGBD que existen en el mercado actual, como PostgresSQL (que viene por defecto instalado y preparado en Mac OSX), SQLite o Microsoft SQL Server.

6.4.1 Descripción y características

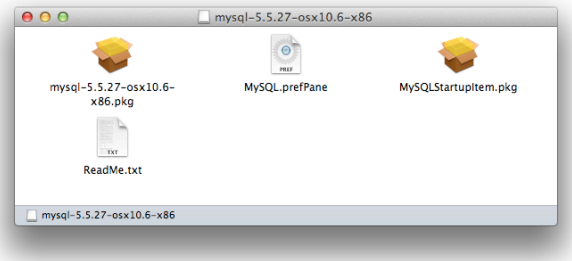
MySQL es un Sistema de Gestión de Bases de Datos relacionales (SGBD), multihilo y multiusuario ampliamente utilizado tanto en el ámbito profesional como particular. La empresa encargada de su desarrollo es "***Sun Microsystems***" empresa que a su vez pertenece a la empresa "***Oracle***". Se distribuye, al igual que Apache, bajo la licencia **GNU GPL,** de la que ya hemos hablado con anterioridad. Pero a diferencia de Apache, MySQL está realizado por una empresa privada y tiene derechos de autor sobre la mayor parte de su código. Ofreciendo así la propia empresa la venta de licencias privativas, soporte y servicio. Su última versión estable es la 5.5.27, mientras que existe una versión en pruebas (beta) que corresponde a la 5.6.5. La versión que utilizamos nosotros durante este proyecto es la 5.5.27 y se puede consultar introduciendo el siguiente comando en el terminal tras su correcta instalación: `> mysql -v`

6.4.2 Instalación y configuración

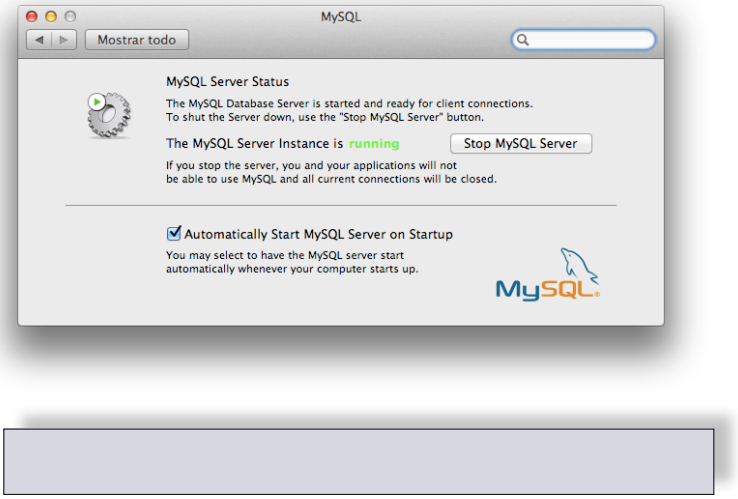
Instalación:

La instalación de este servidor de base de datos en Mac OSX es relativamente sencilla y la detallaremos a continuación:

En primer lugar deberemos de descargar la versión de "*MySQL Community Server*" correspondiente a nuestro sistema operativo en la página oficial del proyecto MySQL del siguiente enlace oficial [31], en nuestro caso descargamos la versión del SO Mac OSX y se nos inicializará la descarga de un módulo con formato .dmg con el siguiente contenido:



El orden de instalación es el siguiente: primero debemos instalar el paquete "*mysql-5.5.27-osx10.6-x86.pkg"* para instalar el demonio *mysql*. Después, opcionalmente, podemos instalar el icono para el panel de preferencias del sistema "*MySQL.prefPane*" para poder iniciar/detener más cómodamente (de manera gráfica) el servidor de base de datos. Tal y como podemos observar en la siguiente captura de dicho panel:



Configuración:

Una vez realizada la instalación básica, tenemos que realizar unos cuantos pasos previos de configuración para dejar el sistema listo para poder almacenar, leer o modificar datos en las tablas. El programa **mysql** se habrá instalado en el siguiente directorio: /usr/local/mysql/bin/mysql. Para poder ejecutar el programa desde un terminal sin necesidad de escribir la ruta completa, debemos añadir este programa a la variable PATH ($PATH) por defecto del sistema, utilizando para ello la siguiente secuencia de comandos en el terminal:

```
> echo $PATH
> /usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/X11/bin
> nano .bash_profile
```

- Añadimos al fichero la línea: `export PATH="/usr/local/mysql/bin:$PATH"`

Lo que hacemos con estas líneas es editar el fichero oculto *.bash_profile (*que se encuentra en la carpeta HOME del usuario) concatenando la variable $PATH a la ruta del ejecutable **mysql**, inidcando así a la consola cuál es el 'PATH' donde puede buscar programas ejecutables, que incluirá ahora la ruta del programa mysql. Una vez llegados a este punto, lo único que nos quedará es definir la contraseña para un usuario administrador "root" en el SGBD, utilizando el siguiente comando:

```
> mysqladmin -u root password NUEVACONTRASEÑA
```

Una vez hecho esto entramos en el sistema identificándonos como el usuario root mediante el comando:

`> mysql –u root –p` (la opción –u indica usuario, y –p hace que nos solicite el password, para más información consultar el manual de mysql [32])

y le concedemos todos los permisos para todas las tablas del sistema mediante la orden:

```
> mysql> grant all on *.* to 'root'@'%' identified by 'NUEVACONTRASEÑA';
```

En este punto esto sólo nos quedará modificar el fichero de configuración de MySQL en la ruta **/etc/my.cnf** para que acepte conexiones por un puerto y socket, para ello insertaremos el siguiente fragmento de código en el apartado [client]:

```
[client]
user        = root
password    = NUEVACONTRASEÑA
port        = 3306
socket      = /tmp/mysql.sock
```

Con toda esta configuración ya tendremos a MySQL listo para utilizar, y empezar a crear y manipular tablas. Es recomendable utilizar un programa que proporcione una interfaz gráfica del SGBD y nos deje interactuar con las tablas. En este proyecto se ha utilizado el software **SequelPro,** del cual se habla más adelante en el apartado de software adicional del anexo.

Con todo esto trabajo realizado, ya tenemos la infraestructura de nuestro servidor totalmente operativa, lista para comenzar a utilizar. Estará todo preparado y configurado: tanto las bases de datos para albergar objetos, como el servidor web necesario para establecer la comunicación con el cliente, utilizando JSON para la codificación de datos, y utilizando scripts PHP para escribir estos datos en la web que leerá dicho cliente, es decir, la aplicación desde el iPhone.

6.5 PHP DAO Generator

6.5.1 Descripción y utilidad

PHP DAO Generator (DAO) [33] proporciona una interfaz abstracta (métodos y clases necesarios) de acceso a una base de datos a través de PHP. Permite a los desarrolladores ejecutar consultas comunes sobre una base de datos sin que tengan que conocer el esquema de tablas de dichas bases. **DAO Generator** separa la lógica de negocio y de datos de una aplicación. Ésta separación es importante ya que es la base de la programación de software multicapa.

El generador de DAO para PHP y MySQL automatiza el proceso de creación de clases de acceso a las diferentes tablas que componen el esquema de nuestro servidor de bases de datos. Podremos acceder a la base de datos a utilizando scripts PHP que invoquen los métodos implementados en estas clases generadas por DAO Generator.

Para la utilización de esta herramienta hay que, en primer lugar, descargarse la última versión de DAO desde su página web, descomprimir el .zip y copiar el contenido en nuestro servidor Apache. Después deberemos Introducir el nombre (dirección) de la base de datos a conectar y contraseña de usuario root en el fichero **templates/class/dao/sql/ConnectionProperty.class.php.** Por último, ejecutamos el script que se encuentra en nuestro caso en la siguiente ruta: http://localhost/php/phpdao/generate.php y ya tendremos las clases -con sus respectivos métodos de acceso a la base de datos- creadas en el directorio **generated** listas para ser utilizadas. A continuación mostramos una captura de ejemplo de utilización de uno de estos métodos que lista todas las filas de la tabla Condiciones e imprime la codificación JSON de la respuesta. Podemos observar que para poder utilizar estos métodos generados hay que incluir el fichero **include_dao.php**, que consiste en la interfaz de las clases que implementan dichos métodos**.** En la parte derecha de la captura se muestra el código del método QueryAll() al que se invoca, cuya implementación se encuentra en la clase creada por esta herramienta para la tabla condiciones *CondicionesMySqlDao.class.php*:

```php
<?php
require_once('include_dao.php');
$transaction = new Transaction();
$todasCondiciones = DAOFactory::getCondicionesDAO()->queryAll();
$transaction->commit();
echo json_encode($todasCondiciones);
?>
```

```php
public function queryAll(){
    $sql = 'SELECT * FROM Condiciones';
    $sqlQuery = new SqlQuery($sql);
    return $this->getList($sqlQuery);
}
```

Figura 53. Captura de pantalla de **Visual JSON**

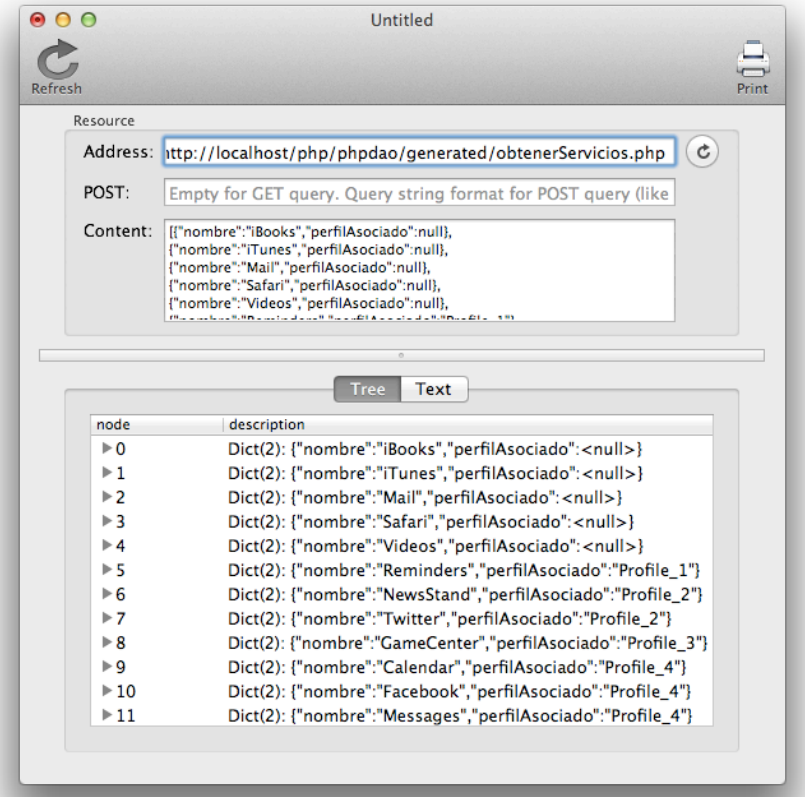
6.6 Software adicional utilizado

En este último apartado del capítulo analizaremos el software que nos ha resultado útil en la realización de este proyecto pero no es imprescindible para el correcto funcionamiento del servidor, por lo que no encaja en ninguno de los apartados anteriores. En concreto hablaremos de los programas Visual JSON y Sequel PRO.

6.6.1 Visual JSON

Visual JSON nos permite visualizar las respuestas formateadas en JSON de la dirección web que le indiquemos. Nos muestra los objetos JSON devueltos a partir de una determinada URL de una manera clara y organizada. La aplicación se puede descargar de manera gratuita de la Mac App Store [34] y está creado por el desarrollador "3rddev". Es una herramienta muy útil para programadores que manejen datos codificados mediante JSON. A continuación podemos ver una captura de esta aplicación donde se pueden apreciar los campos de la interfaz arriba comentados:

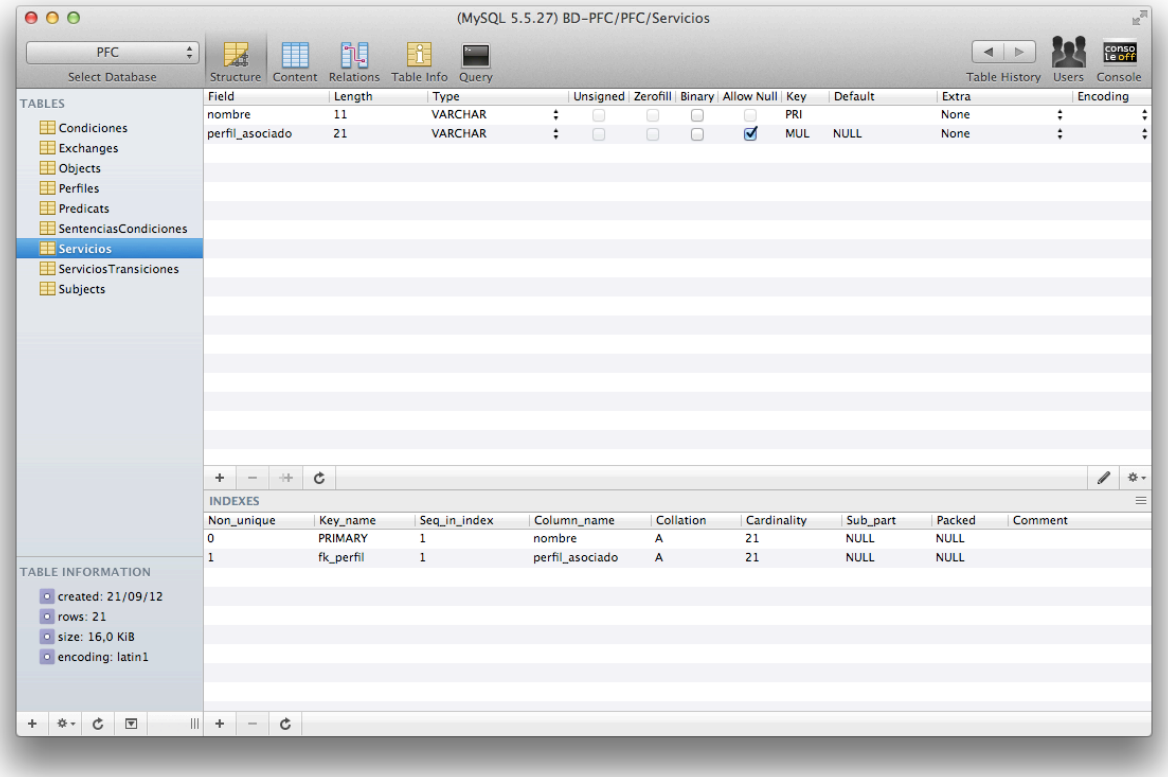
Figura 23. Captura de pantalla de **Visual JSON**



6.6.2 Sequel PRO

SequelPRO [35] es un software gratuito y libre, creado con el apoyo de la comunidad de desarrolladores, cuyo código fuente podemos descargar de desde su propia página web. La versión actual es la 0.9.9.1 y es la que usaremos en el presente proyecto. Este programa nos ofrece una interfaz gráfica formada por todas las bases de datos y tablas que tenga nuestro servidor MySQL. Permitiendo administrar este contenido, es decir, podemos crear/borrar tablas, añadir/eliminar elementos de éstas y modificarlas a nuestro antojo, pudiendo definir restricciones de integridad como claves primarias, claves ajenas, etc. En definitiva nos permite tener toda la potencia de la consola SQL con una atractiva y amigable interfaz gráfica donde se nos muestran todos los detalles de nuestro SGBD. También incluye una consola para ejecutar comando SQL sobre la base de datos de manera tradicional.

A continuación dejamos una captura de dicho software:



Referencias

[1]. http://es.wikipedia.org/wiki/Android#Arquitectura **12**

[2]. http://www.migueldiazrubio.com/2011/12/30/desarrollo-ios-tipos-de-licencias-de-desarrollo/ **15**

[3]. http://es.wikipedia.org/wiki/Darwin_BSD **15**

[4]. http://es.wikipedia.org/wiki/IOS_(sistema_operativo) **16**

[5]. http://es.wikipedia.org/wiki/Framework **17**

[6]. https://www.icloud.com/ **20**

[7]. http://es.wikipedia.org/wiki/Retina_Display **22**

[8]. http://en.wikipedia.org/wiki/Sina_Weibo **27**

[9]. http://es.wikipedia.org/wiki/Smalltalk **29**

[10]. http://es.wikipedia.org/wiki/Gcc **29**

[11]. http://es.wikipedia.org/wiki/Gcc#Lenguajes **29**

[12]. http://es.wikipedia.org/wiki/Runtime **30**

[13]. http://en.wikipedia.org/wiki/Interface_Builder **30**

[14]. http://cocoadevcentral.com/articles/000080.php **31**

[15]. http://developer.apple.com/library/ios/#documentation/uikit/reference/UIViewController_Class/Reference/Reference.html **36**

[16]. http://developer.apple.com/library/ios/#documentation/UIKit/Reference/UITableViewDelegate_Protocol/Reference/Reference.html **38**

[17].http://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSArray_Class/NSArray.html.....39

[18].https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSMutableArray_Class/Reference/Reference.html.....39

[19].https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSPointerArray_Class/Introduction/Introduction.html 39

[20].https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSDictionary_Class/Reference/Reference.html.....40

[21].https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSMutableDictionary_Class/Reference/Reference.html 40

[22].https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/NSMapTable_class/Reference/NSMapTable.html.....40

[23].https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSObject_Class/Reference/Reference.html.....40

[24]. http://es.wikipedia.org/wiki/Apple_Inc.. 42

[25]. http://es.wikipedia.org/wiki/Mac_App_Store 42

[26]. http://es.wikipedia.org/wiki/Servidor_HTTP_Apache.....106

[27]. http://es.wikipedia.org/wiki/GPL.....106

[28]. http://www.kubuntu.org/.....107

[29]. http://json.org/.....110

[30]. http://es.wikipedia.org/wiki/Javascript.....110

[31].http://dev.mysql.com/downloads/mysql/#downloads.....116

[32].http://www.manpagez.com/man/1/mysql/.....118

Bibliografía y Fuentes

Bibliografía:

Objective-C Curso práctico para programadores Mac OSX. iPhone y iPad……
http://www.amazon.es/Objective-c-curso-practico-programadores-iphone/dp/8493831271

Learn Objective-C on the Mac……...
http://www.amazon.es/Learn-Objective-C-Mac-Mark-Dalrymple/dp/1430218150/ref=sr_1_12?s=foreign-books&ie=UTF8&qid=1350837380&sr=1-12

Beginning iOS5 Development – Exploring iOS SDK……..
http://www.amazon.es/Beginning-IOS-Development-Exploring-SDK/dp/1430236051/ref=sr_1_cc_1?s=aps&ie=UTF8&qid=1350837285&sr=1-1-catcorr

Fuentes:

iOS Developer Library……
http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html

iOS Technology Overview……..
http://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iPhoneOSTechOverview.pdf